Projet Informatique

Sujet 3 :
Recherche d'un arbre couvrant minimal par triangulation de Delaunay

Alexandre GRAMFORT, Clément MIGLIETTI

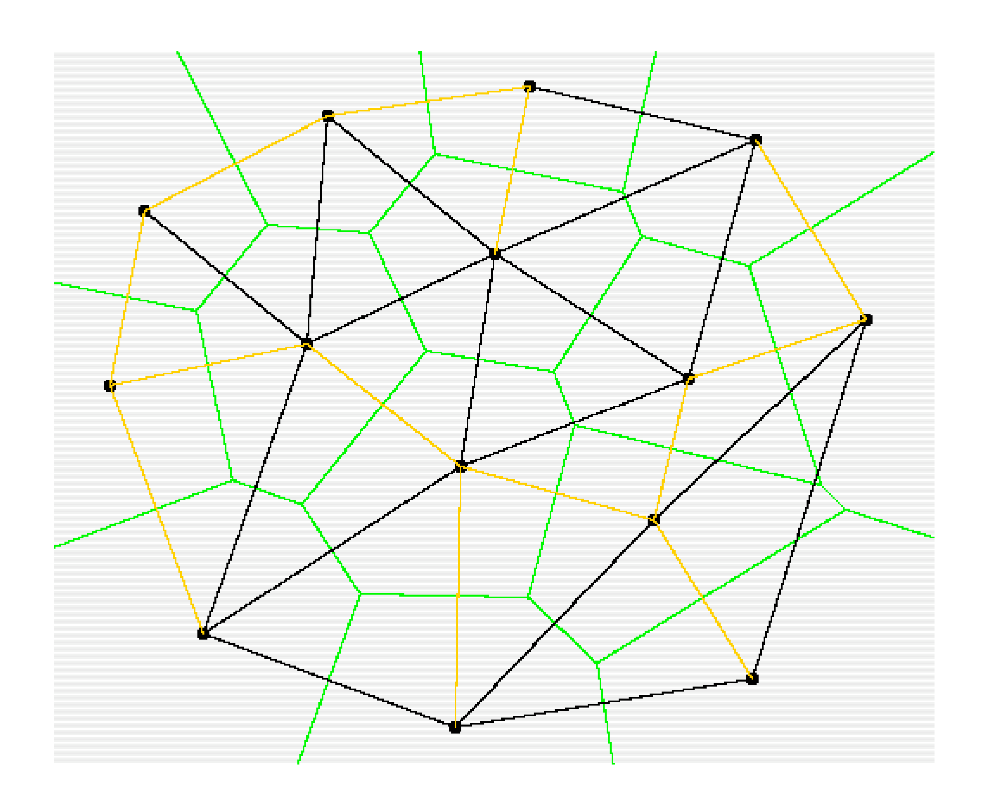


Table des matières :

<u>Utilisation du programme</u>

Le programme a été conçu pour être utilisé de deux façons : la première est d'utiliser l'interface graphique et de rentrer les points à la souris, la seconde de rentrer les points de la triangulation sous forme d'un fichier texte. Dans la fenêtre graphique, un menu permet d'afficher l'arbre couvrant minimal, le diagramme de Voronoï et les cercles circonscrits aux triangles de la triangulation.

Pour plus de détails sur l'utilisation du programme se référer à l'annexe 1.

I. Description des algorithmes utilisés

La construction de l'arbre couvrant minimal se fait en deux étapes. Tout d'abord, on calcule de façon incrémentale la triangulation de Delaunay des points considérés. Puis dans un second temps, à partir des arêtes de la triangulation, on détermine l'arbre couvrant minimal.

1. La triangulation de Delaunay par l'algorithme incrémental

Comme son nom l'indique, le principe de cet algorithme est d'ajouter un à un les points et de modifier en conséquence la triangulation. Dans ce but, nous utilisons la propriété caractéristique de la triangulation de Delaunay : « le cercle circonscrit à un triangle ne contient aucun autre point de la triangulation que ceux du triangle lui-même ». Les modifications sont donc opérées de façon à conserver cette propriété.

a. Principe

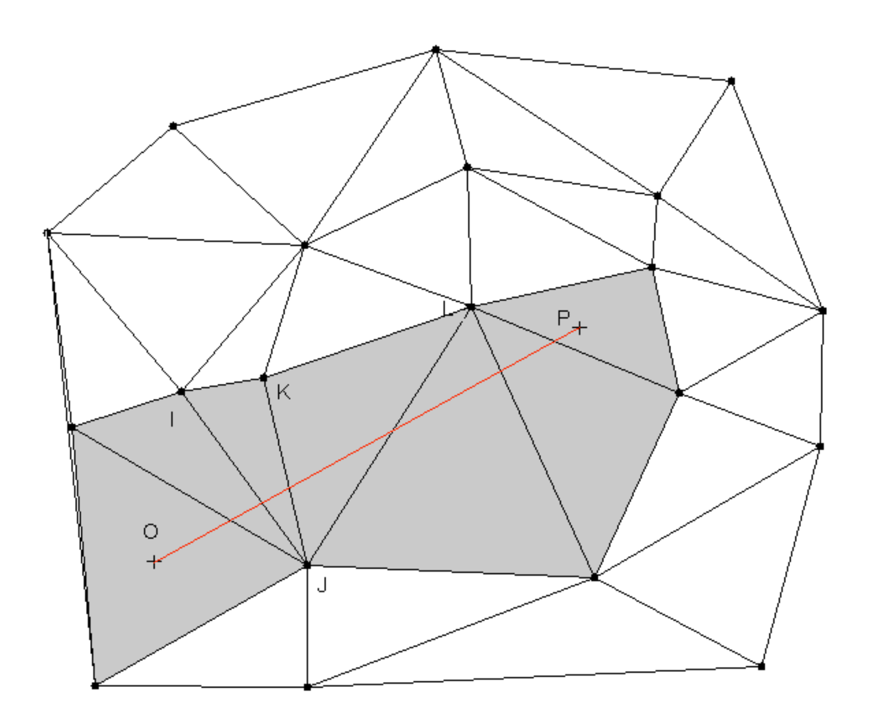
Nous allons détailler le mécanisme de l'insertion d'un point P :

Deux cas peuvent se présenter lors de l'insertion d'un point P : ou bien P est dans un triangle déjà construit, ou bien P est hors de l'enveloppe convexe des points déjà insérés. Pour simplifier l'algorithme, il est possible de ne considérer que le premier cas en considérant que tous les points sont à insérer dans un grand triangle qui sera en fait construit à l'initialisation de la triangulation et avant toute insertion. Cependant, il faut alors remarquer que la triangulation obtenue *in fine* peut ne pas exactement être la triangulation de Delaunay pour les points utilisés en entrée puisque la présence du grand triangle entraînera certaines modifications (nous étudierons plus précisément le moyen d'y remédier en troisième partie). *L'algorithme fournit ici donne donc une solution qui coïncidera avec la solution optimale dans de nombreux cas et qui en fournira une approximation (relativement bonne) sinon.*

Lors de l'insertion d'un point, il s'agit tout d'abord de déterminer la partie de la triangulation qui est constituée des triangles dont les cercles circonscrits contiennent le nouveau point P. Pour localiser un triangle de cette partie, on effectue ce qu'on appelle une localisation par marche.

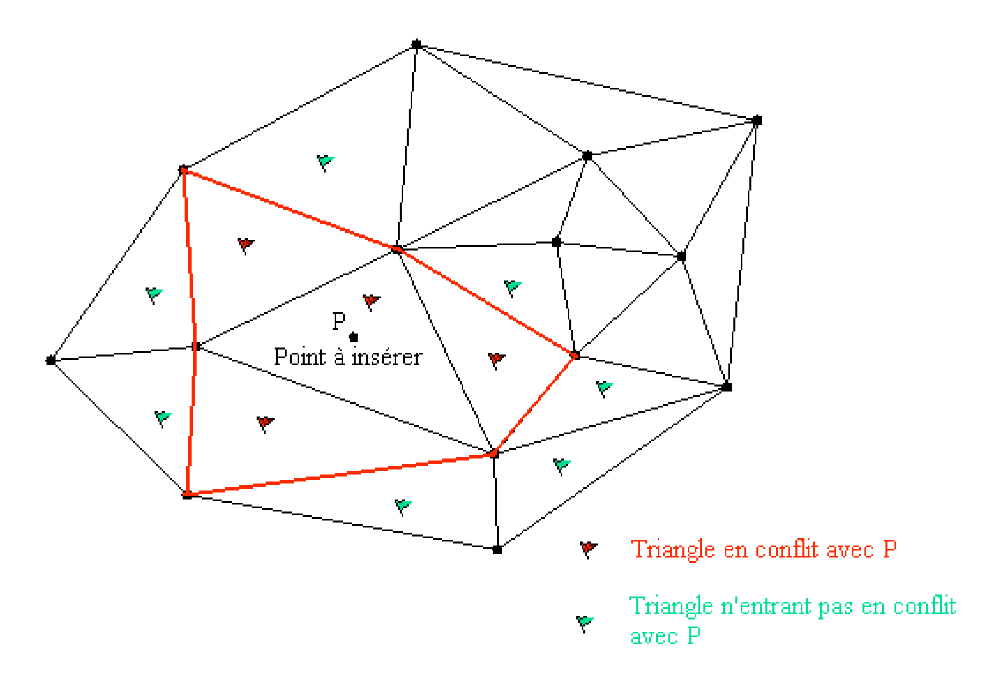
Localisation par marche

Une fois choisi un point O du plan dont la seule contrainte est d'être à l'intérieur d'un triangle de la triangulation, on « se déplace » dans celle-ci grâce aux relations d'adjacences (On a choisi de prendre pour O le centre de gravité d'un triangle connu de la triangulation). Pour passer d'un triangle à un autre nous avons utilisé des conditions d'orientation. Sur la figure, pour sortir de IJK, nous utilisons le fait que OPK et OPJ sont d'orientations contraires. Le triangle suivant est donc le voisin avec lequel IJK partage l'arête KJ.



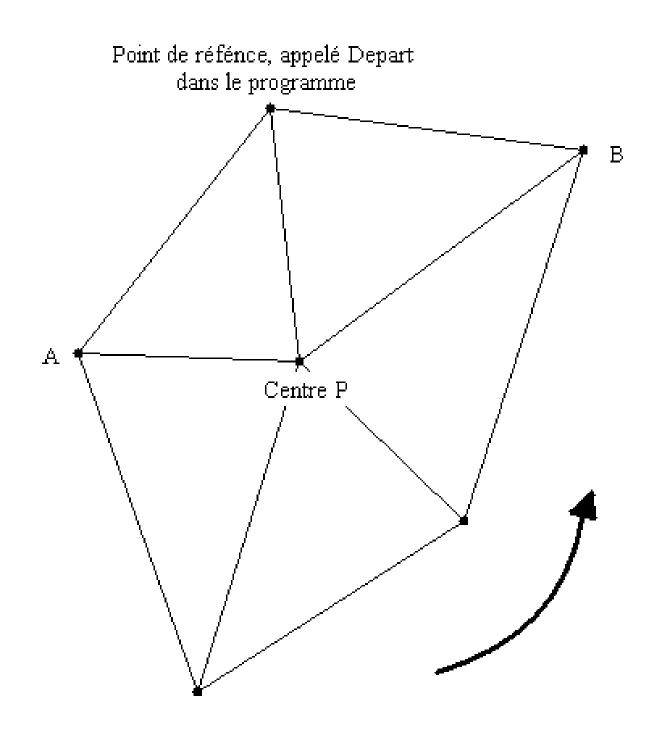
Extraction de la cavité

Une fois la localisation faite, nous connaissons le triangle auquel P appartient. Pour trouver tous les triangles dont les cercles circonscrits contiennent P, on procède à un parcours en profondeur du graphe associé à la triangulation à l'aide des relations d'adjacence. En effet, lorsqu'il s'agit de parcourir la triangulation (par exemple, lorsqu'on veut la dessiner), on peut l'assimiler à un graphe non orienté dont chaque sommet possède au plus trois voisins et donc utiliser un algorithme de parcours en profondeur. Au cours de ce parcours, on marque par « vu » les triangles visités afin de ne pas boucler. La descente en profondeur s'arrête dès qu'on rencontre un triangle pour lequel P n'appartient pas au cercle circonscrit. La cavité est définie par une liste d'arêtes (cf. classe Cavite partie II). Lorsqu'on se trouve dans un triangle, on choisit d'ajouter un segment à la cavité si ce segment se trouve être une arête mitoyenne entre un triangle « en conflit » avec P et entre un triangle ne l'étant pas (le triangle « null » n'est jamais considéré en conflit avec un point à insérer).



Sur la figure, les triangles non-marqués par un drapeau n'ont pas été visités : en effet, ils ne sont pas voisins d'un triangle en conflit avec P et le parcours en profondeur ne les a pas atteint.

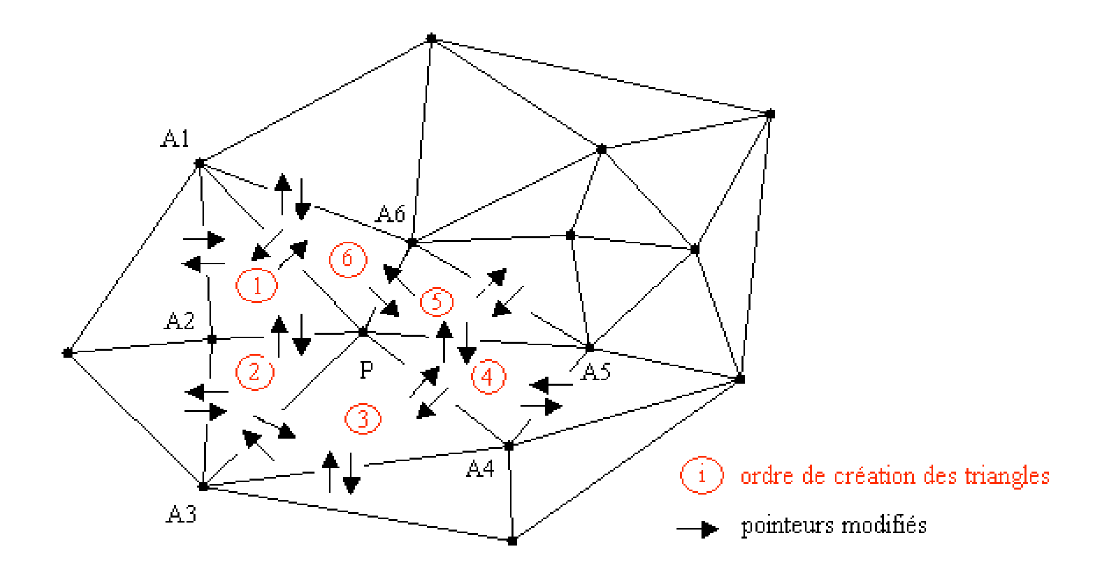
Une fois la cavité exhibée, il faut reconstruire la triangulation en maintenant à jour les relations d'adjacences des triangles. Dans ce but, il est nécessaire de trier les arêtes de la cavité. On a donc défini une relation d'ordre simple.



Avec les notations de la figure, on dira que A est plus petit que B car lorsqu'on parcourt le polygone dans le sens direct depuis le point « Départ », on passe d'abord par A puis par B.

La modification des triangles de la cavité

On peut désormais créer les nouveaux triangles et faire la mise à jour des relations d'adjacence (autrement dit des pointeurs de tous les triangles concernés).

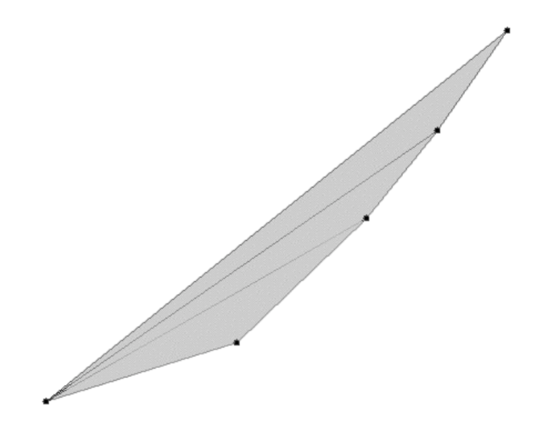


On crée donc les triangles PA_iA_{i+1}, dans l'ordre dans lequel les segments A_iA_{i+1} apparaissent dans la cavité. Les pointeurs rentrant ou sortant des triangles extérieurs à la cavité sont modifiés grâce à un maintien en mémoire pour chaque arête des deux triangles auxquels elle appartenait avant l'insertion.
La mise à jour des pointeurs est véritablement la partie du programme la plus délicate à implémenter.

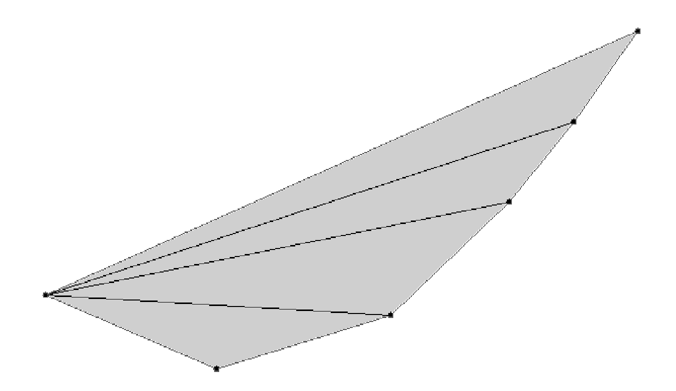
b. Complexité

Nous allons maintenant dire quelques mots sur la complexité du programme. Tout d'abord cherchons une borne supérieure du nombre de triangles créés à chaque insertion. Comme en témoignent les deux graphes suivants, nous voyons qu'il est possible lors de l'insertion du point n+1 de détruire l'ensemble des triangles jusque-là construits et d'en créer alors n-1.

Triangulation après insertion de 5 points :



Triangulation après insertion de 6 points :



Ce cas est clairement le pire et donc le coût à l'étape n+1 et au plus linéaire en nombre de nouveaux triangles créés. On en déduit donc une majoration de la complexité de l'insertion de n points de façon incrémentale : <u>*$O(n^2)$*</u>.
En ce qui concerne la borne inférieure. On peut remarquer qu'en placant comme ci-dessus les points sur une parabole, le fait de trianguler donne un tri des abscisses des points et de fait ne peut avoir une meilleure complexité que *n ln n*. L'algorithme est donc un <u>*□(n ln n)*</u>.

2. L'algorithme de construction de l'arbre minimal

a. Principe

Le principe général de construction est donné dans l'énoncé du sujet. Nous nous contenterons donc de démontrer les lemmes servant à justifier l'utilisation de la triangulation de Delaunay pour la recherche de l'arbre couvrant minimal.

Théorème :
l'arbre couvrant minimale est inclus dans la triangulation de Delaunay.

Nous donnons une démonstration de cette propriété utilisant les lemmes suivants :

Lemme 1 : Si $S = S_1 \cup S_2$ alors la plus courte arête entre un point de S_1 et un point de S_2 est une arête de la triangulation de Delaunay.

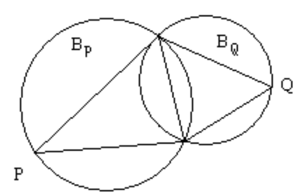
Lemme 2 : Si S = S_1 U S_2, alors la plus courte arête entre un point de S_1 et un point de S_2 est une arête de l'arbre couvrant minimal.

Lemme 3 : Une arête est de Delaunay si et seulement si il existe un cercle passant par ses extrémités en ne contenant pas de point de S.

Lemme 4 : Soit A,B,C,D quatre points du plan. Soit □ un cercle passant par A et B. Si C et D sont extérieurs à □ alors C est extérieur au cercle circonscrit à ABD et D est extérieur au cercle circonscrit à ABC.

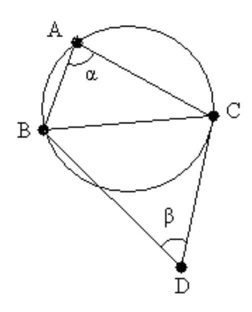
Lemme 5 : Soit ABC et BCD deux triangles adjacents et possédant la propriété de Delaunay. Alors, les triangles ABD et ACD ne possèdent pas la propriété de Delaunay.

Rappelons tout d'abord qu'avec les notations de la figure, on a :



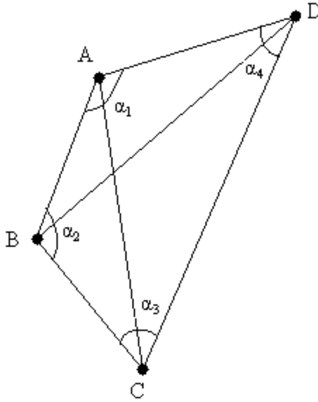
Q n'appartient pas à B_P □ P n'appartient pas à B_Q. (1)

D'autre part, notons que la propriété « D est à l'extérieur du cercle circonscrit à ABC » se traduit dans la configuration du dessin par □<π-□.



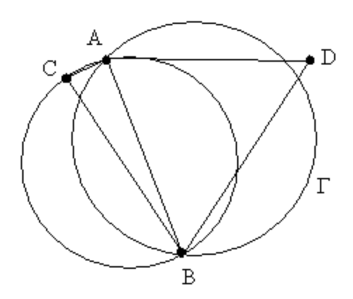
Démonstration du lemme 5 :

Supposons que les 4 triangles possèdent la propriété de Delaunay.
Par ce qui précède on a alors :



$\alpha_3 < \pi - \alpha_1$ et $\alpha_4 < \pi - \alpha_2$ et donc $\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 < 2\pi$. Contradiction.

Démonstration du lemme 4 :



On pose α l'angle tel que Γ soit l'ensemble des points tels que :

$AMB = \alpha$ si M est dans le même demi-plan que C ou tel que

$AMB = \pi - \alpha$ si M est dans le même demi-plan que D.

On pose α_1 l'angle tel que $ACB = \alpha_1$. Le cercle circonscrit à ACB est l'ensemble des points tels que

$AMB = \alpha_1$ si M est dans le même demi-plan que C, ou tels que

$AMB = \pi - \alpha_1$ si M est dans le même demi-plan que D.

C étant extérieur à Γ, on a $\alpha_1 < \alpha$

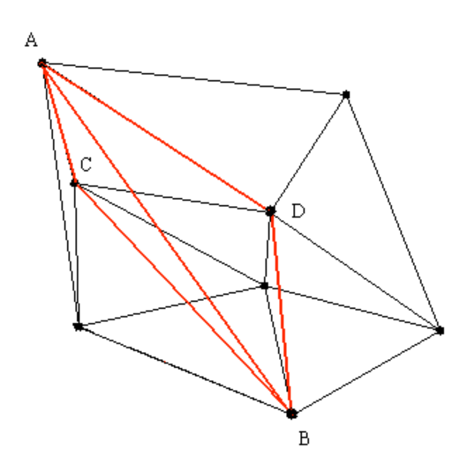
On a alors $ADB < \pi - \alpha < \pi - \alpha_1$ et donc D est extérieur au cercle circonscrit à ACB.

Par (1), on en déduit que C est extérieur au cercle circonscrit à ABD.

Démonstration du lemme 3 :

Le sens indirect résulte de la définition de la triangulation de Delaunay.

Sens direct : considérons une arête [AB] telle qu'il existe un cercle passant par ses extrémités en ne contenant pas de points de S. Supposons qu'elle ne soit pas de Delaunay. Par convexité de la triangulation, elle coupe un segment [CD] tel que ACD appartienne à la triangulation.



Du lemme 4, on déduit que ABC et ABD possèdent la propriété de Delaunay. Or ACD est un triangle de la triangulation. Il possède donc lui aussi cette propriété, ce qui contredit le lemme 5. L'arête [AB] est donc de Delaunay. D'où le lemme 3.

Démonstration du lemme 2 :

Soit [AB] la plus courte arête d'un point de S_1 à un point de S_2.
Considérons □ l'arbre couvrant minimal de S. Si [AB] n'appartient pas à □, en remplaçant une arête de □ qui relie un point de S_1 à un point de S_2 par [AB], on obtient un nouvel arbre couvrant mais plus court, ce qui est contradictoire avec le fait que □ soit l'arbre couvrant minimal. D'où le résultat.

Démonstration du lemme 1 :

Soit [AB] la plus courte arête entre un point de S_1 et un point de S_2 (A dans S_1, B dans S_2). Soit □ le cercle de diamètre [AB]. On suppose qu'il existe un point P intérieur à □.
Si P est dans S_1, alors $PB < AB$, ce qui est contradictoire,
Si P est dans S_2, alors $PA < AB$, ce qui est aussi contradictoire.
Donc □ est vide. Donc par le lemme 3, on conclu que [AB] est de Delaunay.

Démonstration du théorème :

Considérons □ l'arbre couvrant minimal de S. Soit [AB] une de ses arêtes. □ \ [AB] sépare S en deux composantes connexes que l'on nomme S_1 et S_2. [AB] est nécessairement la plus courte arête entre S1 et S2 (il ne peut y en avoir qu'une dans □ par minimalité de □, et le lemme 2 permet d'affirmer que c'est la plus courte). Par le lemme 1, [AB] est de Delaunay. D'où le résultat.

Le théorème étant acquis, on en déduit l'algorithme de construction proposé dans le sujet.

Détails sur la structure de données utilisée

Nous avons envisagé la classe Graphe comme complétement indépendante de la triangulation dans la mesure où, elle peut aussi être utilisée sans Delaunay en mettant dans le graphe initial toutes les arêtes possibles entre chaque point. La

classe Graphe possède donc un tableau de points qui n’est rien d’autre qu’une copie des points de la classe Triangle. On stocke les relations d’adjacences du graphe dans un tableau contenant en position i une liste d’entiers correspondant aux indices des voisins de i. Ceci permet, notamment lors de la recherche des voisins de i lors de l’ajout de i à l’arbre couvrant minimal, de ne pas tester si i est relié à tous les autres points comme cela aurait été nécessaire avec un stockage sous forme de matrice. La queue, elle, possède un tableau s1 de boolean, comme le suggère l’énoncé, et une liste triée de segments autorisés pour l’ajout du point suivant. Le fait que cette liste soit triée de façon croissante facilite grandement la recherche du segment minimum : c’est toujours le premier élément de la liste.

b. Complexité

Ce qui détermine la complexité de l’algorithme de construction d’arbre minimal est la structure de donnée utilisée pour la queue. Avec une liste triée, au pire des cas, on ajoute à la liste lors de l’étape n un segment en fin de liste. On a alors dans ce cas un coût lineaire en n pour passer de n à n+1. La complexité finale est donc en $O(n^2)$. L’utilisation d’un arbre bianire équilibré pour la gestion de la queue aurait permis une recherche de minimum en *ln n* et une complexité finale en *O(n ln n)*. Cependant l’algorithme de Delaunay étant en $O(n^2)$ ceci n’est pas grave. Il aurait été intéressant d’implémenter une telle structure de données si l’algorithme de triangulation par division-fusion avait été choisi. La complexité globale aurait alors été en *O(n ln n)*.

II. Description de l'architecture du programme

Le programme est structuré en un certain nombre de classes où l'on peut distinguer trois groupes : les classes de l'algorithme de triangulation, celles utilisées pour la détermination de l'arbre de recouvrement minimal, et enfin les classes ayant servi à l'interface graphique et à l'entrée des points sous forme de fichier texte.

D'autre part, nous avons adopté un style de programmation plutôt orienté objet pour traiter le sujet.

1. Les classes de la triangulation de Delaunay

Remarques générales sur l'implémentation

Nous avons choisi de stocker tous les points figurant dans la triangulation dans le tableau static *points* de la classe Triangle. La quasi-totalité des méthodes du programme ne prend donc pas en arguments ces points mais des entiers qui représentent leurs indices dans le tableau. Mentionnons également que les trois premières cases de ce tableau sont occupées par les trois points qui définissent le grand triangle que nous avons évoqué plus haut. Nous avions, pour faciliter l'écriture du programme, pensé mettre le tableau de points dans une classe dont toutes les autres classes hériteraient mais Java ne permet pas l'héritage multiple et malheureusement certaines classes graphiques dérivent déjà de classes d'AWT. Nous avons donc préféré la solution finalement adoptée.
D'autre part, la taille du tableau *points* étant fixée arbitrairement à l'avance, nous avons utilisé une variable statique *nb* tel que *nb + 3* soit l'indice du dernier point inséré (autrement dit le nombre de points déjà insérés vaut *nb+1*). Cet entier sert à trianguler au fur et à mesure de l'ajout des points à la souris par le biais de l'interface graphique.

La classe Point

Un point est simplement représenté par un couple de double. Nous aurions pu choisir de mettre des entiers ou d'utiliser la classe Point d'AWT mais les erreurs numériques d'arrondi auraient alors été pénalisantes notamment pour la vérification de certains prédicats géométriques. Cette remarque est encore plus pertinente lorsque les points sont fournis pas fichier texte dans la mesure où dans ce cas abscisse et ordonnée peuvent être des doubles. Avec l'interface graphique, la position de la souris est décrite par deux entiers. Il convient donc alors de convertir ces entiers en doubles.

La classe Triangle

Un triangle est décrit par ses trois sommets (trois indices entiers du tableau de points) ainsi que par des pointeurs vers ses trois triangles voisins (pointeurs

éventuellement *null* dans le cas de triangles étant au bord de la triangulation). Chaque objet de type Triangle comporte également un champ *centre* correspondant au centre du cercle circonscrit au triangle, ainsi que deux champs notés *vu* et *estDessine* qui sont des « drapeaux » (« flag ») utilisés par les fonctions parcourant la triangulation. *vu* sert lors de la triangulation alors que *estDessine* sert pour l'affichage graphique.

La classe Triangle a été munie de méthodes servant à vérifier certains prédicats géométriques :

- *retourneCentre()* qui renvoie le centre du cercle circonscrit au triangle

Le coordonnées du centre du cercle circonscrit au triangle ABC sont données par les expressions :

$$x_O = \frac{1}{2}\,\frac{\begin{vmatrix} z_2 & y_2 \\ z_3 & y_3 \end{vmatrix}}{\begin{vmatrix} x_2 & y_2 \\ x_3 & y_3 \end{vmatrix}} + x_A \;;\quad y_O = \frac{1}{2}\,\frac{\begin{vmatrix} x_2 & z_2 \\ x_3 & z_3 \end{vmatrix}}{\begin{vmatrix} x_2 & y_2 \\ x_3 & y_3 \end{vmatrix}} + y_A$$

où x_2, y_2, z_2 (resp. x_3, y_3, z_3) désignent (notations utilisées dans le code) l'abscisse, l'ordonnée et la norme au carré du vecteur **AB** (resp. **AC**).

- *retourneCentreGravite()* qui renvoie le centre de gravité du triangle

  Cette méthode sert pour partir d'un point intérieur à la triangulation lors de la localisation par marche.

- *estOriente()* qui renvoie 1 si le triangle est direct, -1 s'il est indirect, 0 s'il est plat

Pour tester si le triangle ABC est orienté, on calcule le déterminant det(AB,AC). S'il est de signe positif, alors le triangle est orienté, sinon non.

- *dansCercle(int p)* qui renvoie 1 si le point d'indice p dans le tableau *points* est dans le cercle, -1 s'il est à l'extérieur, 0 s'il est dessus. Cette fonction utilise donc le déterminant de cocyclicité ci –dessous :

$$\begin{vmatrix} 1 & x_A & y_A & x_A^2+y_A^2 \\ 1 & x_B & y_B & x_B^2+y_B^2 \\ 1 & x_C & y_C & x_C^2+y_C^2 \\ 1 & x_D & y_D & x_D^2+y_D^2 \end{vmatrix}$$

- *dansTriangle(int p)* qui renvoie *true* si le point d'indice p dans le tableau *points* est intérieur au triangle, *false* sinon.

Pour cela, on se sert de la propriété : « le point P est intérieur au triangle ABC équivaut à PAB, PBC et PCA sont directs ». Cette fonction fait donc trois appels à *estOriente*.

Enfin, une méthode *print()* qui affiche une chaîne de caractères décrivant le triangle (servant essentiellement au débugage) a également été écrite.

La classe Triangulation

Une triangulation est décrite par l'ensemble de ses triangles. On dispose d'un pointeur d'entrée dans la triangulation que nous avons nommé *triangleInitial*. Ce dernier est déplacé à chaque insertion d'un nouveau point. On a muni la classe triangulation des méthodes suivantes :

Tout d'abord les méthodes pour l'insertion :

- *localise(int p)* qui renvoie le triangle qui contient P

Localise effectue une localisation par marche à partir de triangleInitial et prend pour point de départ son centre de gravité.

- *extraitCavite(Triangle t, int p)* qui renvoie la cavité autour du point d'indice p en partant du triangle localisé t
- *remplitCavite(Cavite c, int p)* met alors à jour la triangulation autour de p après avoir trié la cavité.

Le tri utilise la méthode *plusPetit(int a, int b, int centre, int depart)* et la méthode *insereDansCavite(int aa,int bb,Triangle tin,Triangle tex,Cavite cav,int centre)* qui insère en conservant l'ordre définit par *plusPetit*.

Il s'ajoute à cela un certain nombre de méthodes servant à l'exportation postscript ainsi que les méthodes servant à la création du graphe.

- *retourneGraphe()* renvoie le Graphe recherché
- *retourneArbre()* renvoie l'arbre après avoir calculé le graphe.

Quant à la fonction *main*, elle permet d'insérer les points à partir d'un fichier texte. Pour plus de détails voir Annexe 1.

La classe Cavite

La cavité est définie comme étant une liste de segments triés (cf. la relation d'ordre introduite ci-dessus). Deux premiers champs *a* et *b* désignent les extrémités du segment. De plus, à chaque segment sont associés deux pointeurs : un pointeur tInt qui pointe vers le triangle intérieur à la cavité ayant pour arête [ab], un pointeur tExt qui pointe vers le triangle extérieur à la cavité ayant pour arête [ab]. Ces pointeurs servent lors de la création des nouveaux triangles qui vont venir combler la cavité (cf. ci-dessus).

2. Les classes de l’arbre couvrant minimal

La classe Graphe

La classe Graphe stocke les relations d’adjacence dans *voisins* qui est un tableau de listes de points, ainsi que les points dans un tableau nommé *coords*. La matrice de boolean *estDejaAjoute* permet de s’assurer qu’aucune arête n’est ajoutée deux fois à *voisins*.
Les méthodes de la classe Graphe sont :

- *ajouter(int i, int j)* qui comme son nom l’indique, ajoute l’arête (i,j).
- *calculerArbre()* renvoyant l’arbre couvrant minimal.

La classe QueueListe

Cette classe contient un tableau de boolean *s1*¤suggéré par l’énoncé ainsi qu’une liste triée de segments, sans oublier le tableau *coords* de Graphe afin d’être en mesure de préciser la taille des arêtes lors de l’ajout. Une méthode permet l’ajout d’un point d’indice i¤: *ajouter(int i, ListeEntier[] voisins)*. Celle-ci adopte fidèlement l’algorithme de l’énoncé.

La classe ListeSegment

Cette classe n’est rien d’autre qu’une implémentation de liste triée croissante selon la longeur des segments. La méthode ajouter est ici exceptionnellement choisie *static* pour éviter l’embarras des pointeurs *null*.

La classe ListeEntier

C’est une simple liste d’entiers.

La classe Arbre

Cette classe correspond à une liste de Segment.

La classe Segment

Elle contient 3 champs¤: les deux entiers des indices des points avec la distance qui les sépare.

3. Les classes graphiques et la classe de lecture

Sans rentrer dans les détails, voici quelques informations sur l’implémentation de l’interface graphique.

La classe Delaunay

Cette classe dérive de Frame et contient comme champs¤: un CanvasDelaunay¤(*monCanvas*), un PanelDelaunay¤(*monPanel*) ainsi que l’ensemble

du code nécessaire à l'affichage des menus. Elle possède aussi quelques booleans static servant à déterminer les éléments à afficher dans *monCanvas*.

La classe CanvasDelaunay

Cette classe dérive de la classe Canvas d'AWT et constitue la portion de la fenêtre sur laquelle les points sont placés à la souris. Elle implémente donc naturellement MouseListener et possède un KeyAdapter permettant la gestion des racourcis claviers.
Elle possède également toutes les méthodes servant au dessin de la triangulation.

La classe PanelDelaunay

Elle gère le champ-texte et le bouton de bas de fenêtre permettant de modifier le nombre de points pouvant être insérés.

Ces trois méthodes se partagent une même instance de Triangulation.

La classe Lecture

Cette classe permet l'entrée des points sous forme d'un fichier texte. C'est un simple lexeur adapté à l'entrée des points sous forme « abscisse-ordonnée » au format double.

III. Critique des résultats et améliorations de l’algorithme

Le premier constat que nous avons fait après avoir implémenté cette méthode fut que si le triangle initial n’était pas suffisamment grand, il était très facile de n’obtenir qu’une approximation de la triangulation souhaitée. En effet, celle-ci pouvait après suppression des triangles extérieurs ne plus contenir les arêtes de l’enveloppe convexe. Ceci était génant car celles-ci pouvaient être dans l’arbre couvrant minimal. Pour remédier à cela nous avons pensé à une première modification de notre approche.

1. Une première modification

Pour obtenir la triangulation de Delaunay nous avons pensé à :

- commencer par déterminer les points de l’enveloppe convexe des points de la triangulation
- insérer ces points dans le grand triangle
- « détacher » ces points du grand triangle, c’est-à-dire détruire toutes les arêtes touchant une des extrémités du grand triangle, et mettre à *null* les pointeurs des triangles touchant le bord de l’enveloppe convexe
- insérer les autres points de la triangulation, qui désormais seront tous intérieurs à l’enveloppe qui vient d’être créée

Le calcul d’enveloppe convexe ne se fait donc qu’une seule fois et par l’algorithme de Graham (en n ln(n) du nombre de points). Il s’effectue ainsi en travaillant sur les points (et non pas sur les triangles, on ne fait donc pas de parcours en profondeur pour calculer l’enveloppe convexe).
Avec cette méthode, on obtient notamment un diagramme de Voronoï correct au bord de la triangulation.
Nous avons implémenté cette modification en ajoutant la classe suivante :

La classe Enveloppe

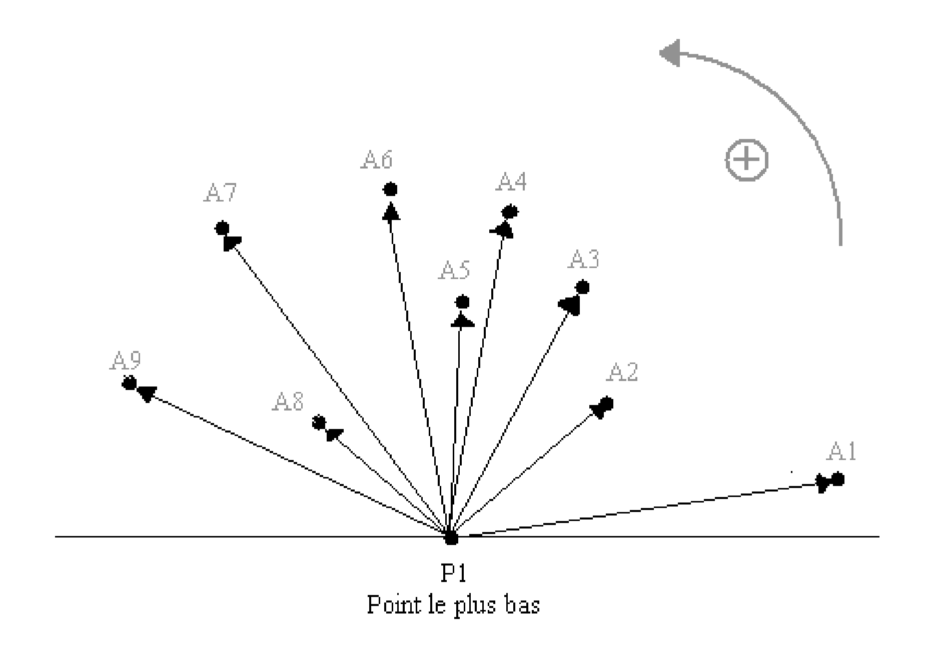
La classe Enveloppe permet la gestion du calcul de l’enveloppe convexe des points sur lesquels va s’appuyer la triangulation. Un objet de type Enveloppe comporte 3 champs :

- Un champ *points* qui ne sera utilisé que comme un pointeur vers le tableau de points de la classe triangle.
- Un int *m* qui donne l’indice du tableau *points* du premier point n’appartenant pas à l’enveloppe convexe
- Un int *nb* qui contient le nombre de points de la triangulation.

La classe Enveloppe est munie des fonctions :

- *theta(point p1, point p2)* qui rend l’angle entre le vecteur P_1P_2 et l’axe des abscisses.

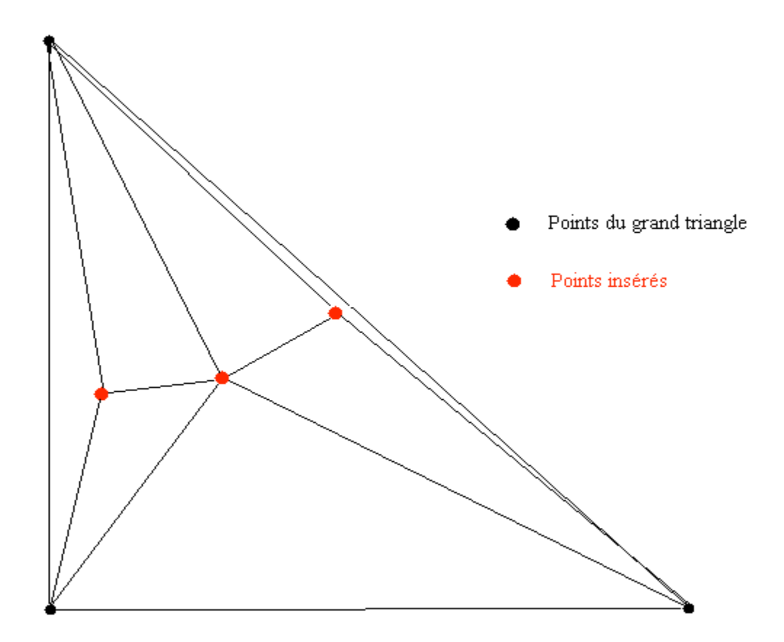
- D'une fonction *tri* qui est un quick sort triant les *nb* points du tableau par ordre des thêta croissant (avec pour points P_1, le point d'abscisse la plus petite).



- ➢ D'une fonction *enveloppeConvexe* qui, à partir de là, détermine l'enveloppe convexe des points en remplissant le champ *m* défini ci-dessus.

Le constructeur fait un appel à la fonction *enveloppeConvexe* et donc modifie le tableau de points qui lui est rentré en argument.

Critique de cette méthode :
Si cette manière de procéder permet d'obtenir la triangulation de Delaunay souvent de manière plus correcte, elle génère cependant des erreurs dans certaines situations, et en particulier lors des premières insertions lorsque les points forment des triangles très aplatis. Les erreurs sont dues au fait que les segments de l'enveloppe convexe peuvent ne pas appartenir à la triangulation. Dans le pire des cas, on peut observer la configuration où tous les triangles formés touchent le grand triangle (cf. dessin). C'est ce cas particulier qui nous a posé problème. Bien sûr dans la pratique, il est relativement peu probable si les points sont répartis à peu près aléatoirement dans la fenêtre graphique et si la taille du grand triangle de départ est augmentée.

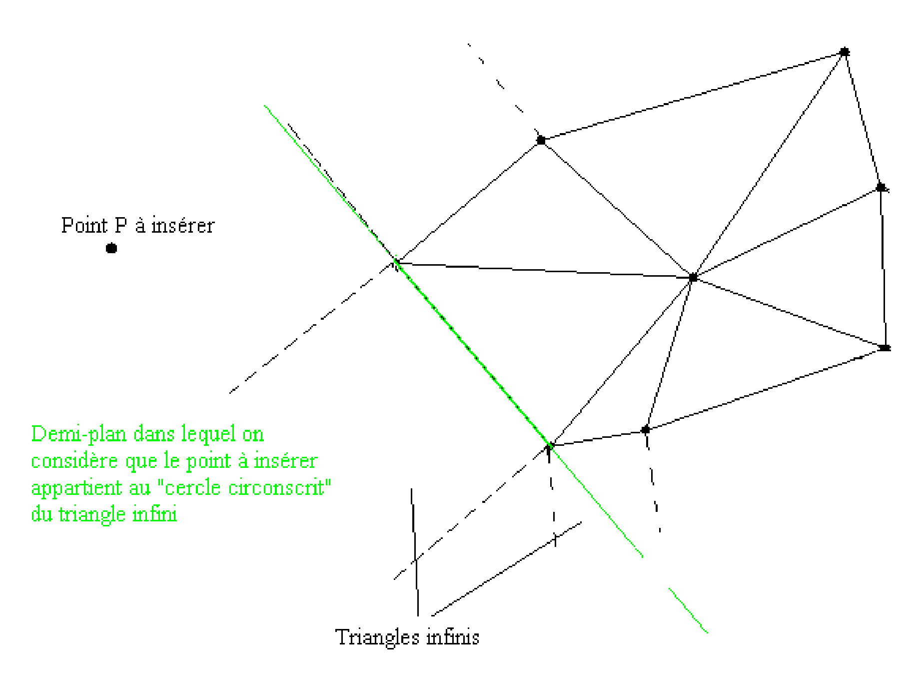


Une fois encore ici le problème est imparfaitement résolu et impose comme seule solution d'augmenter la taille du triangle initial. Cependant une méthode exacte existe.

Remarque : Nous n'avons pas fourni le code de la classe Enveloppe car l'ayant abandonné par constat d'imperfection. Cependant, il reste bien sûr disponible sur votre demande.

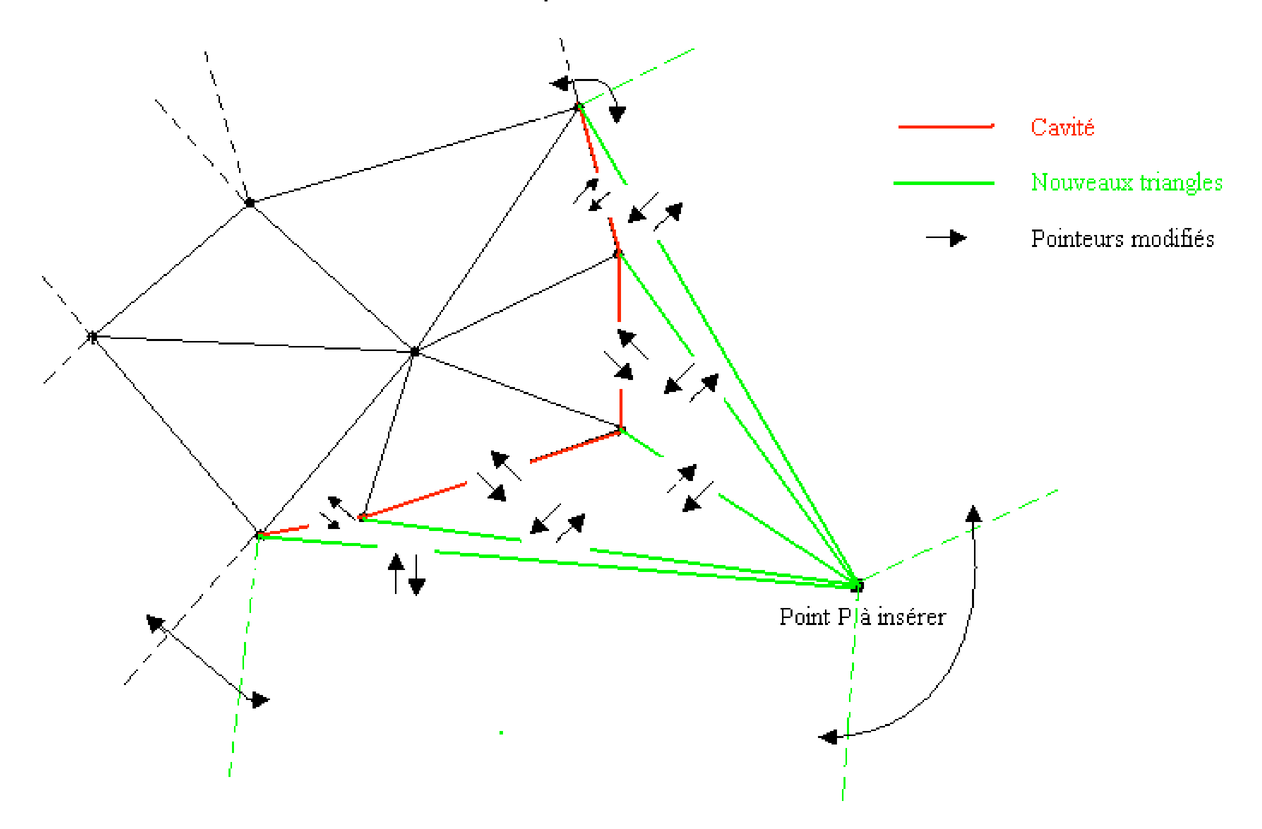
2. Une méthode exacte

Nous présentons enfin une autre façon de procéder qui cette fois fournit une solution exacte au problème posé : la méthode consiste à considérer que les arêtes des bords appartiennent à un triangle dont le troisième point est à l'infini (cf. dessin).



Par manque de temps, nous n'avons pas implémenté cette méthode mais voici les modifications qu'il faudrait opérer :

- On matérialiserait le point infini par l'indice du tableau des points –1.
- Un triangle infini devrait avoir pour triangles adjacents les triangles infinis des bords les plus proches, de façon à pouvoir effectuer un parcours en profondeur pour trouver la cavité.
- Dans la fonction *dansCercle* de la classe triangle, il faudrait ne pas calculer le déterminant si le triangle est de type infini mais vérifier si le point à insérer appartient au demi-plan représentant le « cercle circonscrit » de ce triangle.
- Concernant la construction de la cavité, il faut imposer de ne pas ajouter à la cavité les segments comportant un point infini.
- Un autre problème apparaît dans l'ordonnancement des segments de la cavité : puisque le point de départ (cf. I 1 a) figure) de la suite de segments est pris au hasard, dans la majorité des cas, après tri, nous obtiendrions une suite de segments non contigüs. Ce problème peut être résolu en refaisant un tri avec pour point de départ, le point de rupture (on obtient alors une suite de segments contigüs définissant la cavité).
- Le remplissage de la cavité est également à modifier puisque dans le cas d'une insertion d'un point « extérieur » il faudrait créer deux triangles infinis avec pour pointeur les triangles des bords voisins. La mise à jour des pointeurs est donc assez différente.
- Enfin puisqu'il n'y aurait plus aucun triangle *null*, il faudrait remplacer tous les tests faisant référence au triangle *null* par des tests à l'aide d'une méthode *estInfini* qui renverrait *true* si le triangle considéré est infini, c'est-à-dire si l'un des sommets a pour indice -1.



Bilan

Après ces quelques critiques sur notre algorithme, nous pouvons tout de même considérer que notre implémentation reste valable sachant que pour un ensemble de points donnés, en cas de problème de convexité de la triangulation obtenue, il est toujours possible d'augmenter la taille du triangle initial jusqu'à résolution du problème.

Annexe 1 : Manuel d’utilisation

Il existe deux façon de faire fonctionner le programme.

Première méthode :
La première est de fournir en entrée du programme un fichier texte contenant une liste de points. Chaque ligne de ce fichier contient l’abscisse et l’ordonnée d’un point.

Exemple d’un fichier input.txt :

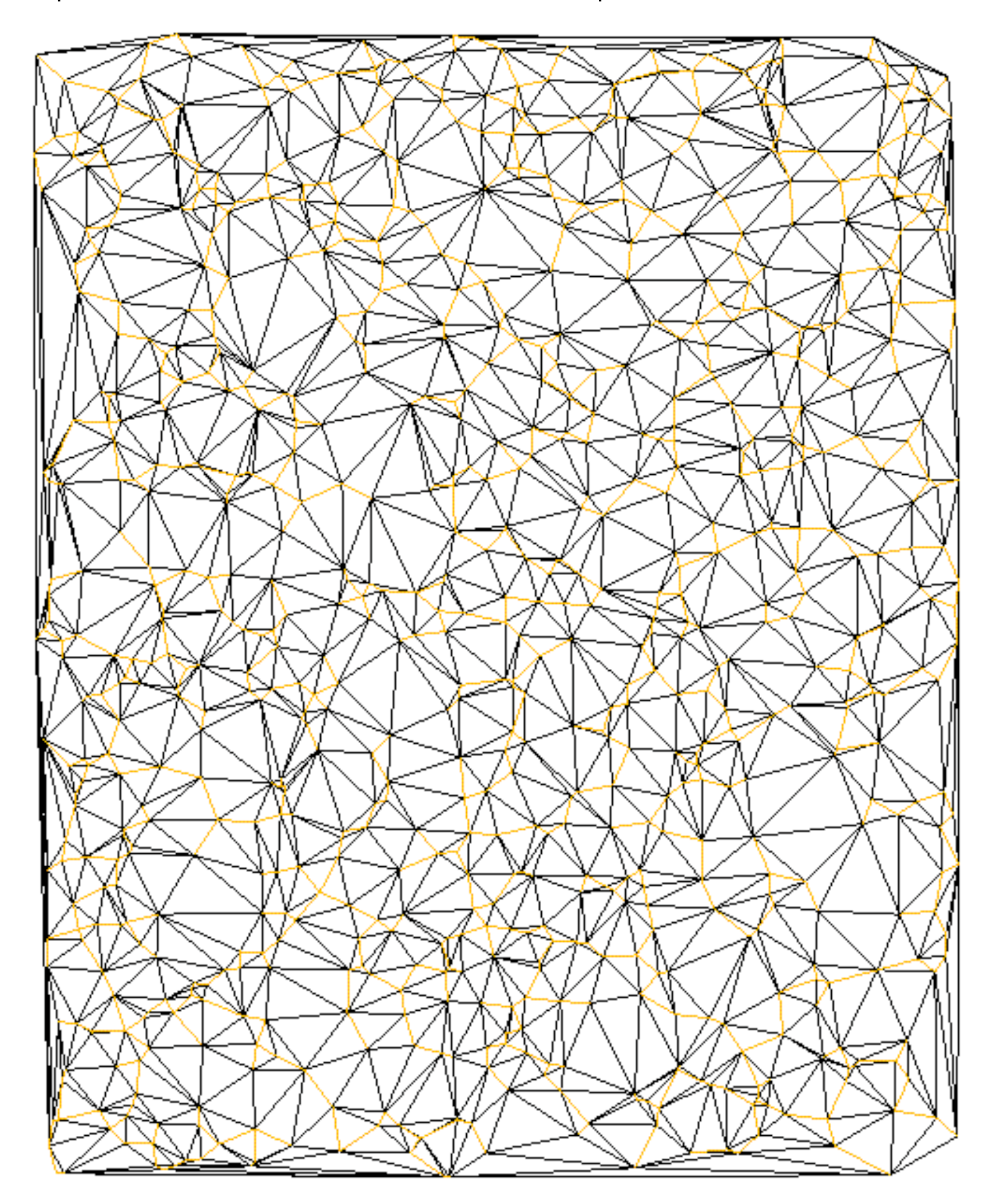
```
62.1155 83.4814
55.0049 377.853
514.752 610.353
239.056 116.934
237.073 560.439
162.64 6 93.529
227.307 395.033
375.592 123.253
…
```

La triangulation s’effectue alors en tapant la ligne de commande :

```
> java Triangulation < input.txt
```

La triangulation est alors écrite en postscript dans un fichier nommé par défaut « output.ps »

Exemple de résultat avec le fichier d'environ 800 points fourni avec le code :



Deuxième méthode :
La seconde est d'utiliser l'interface graphique qui permet de saisir les points à la souris.

Pour l'utiliser, il suffit de taper :

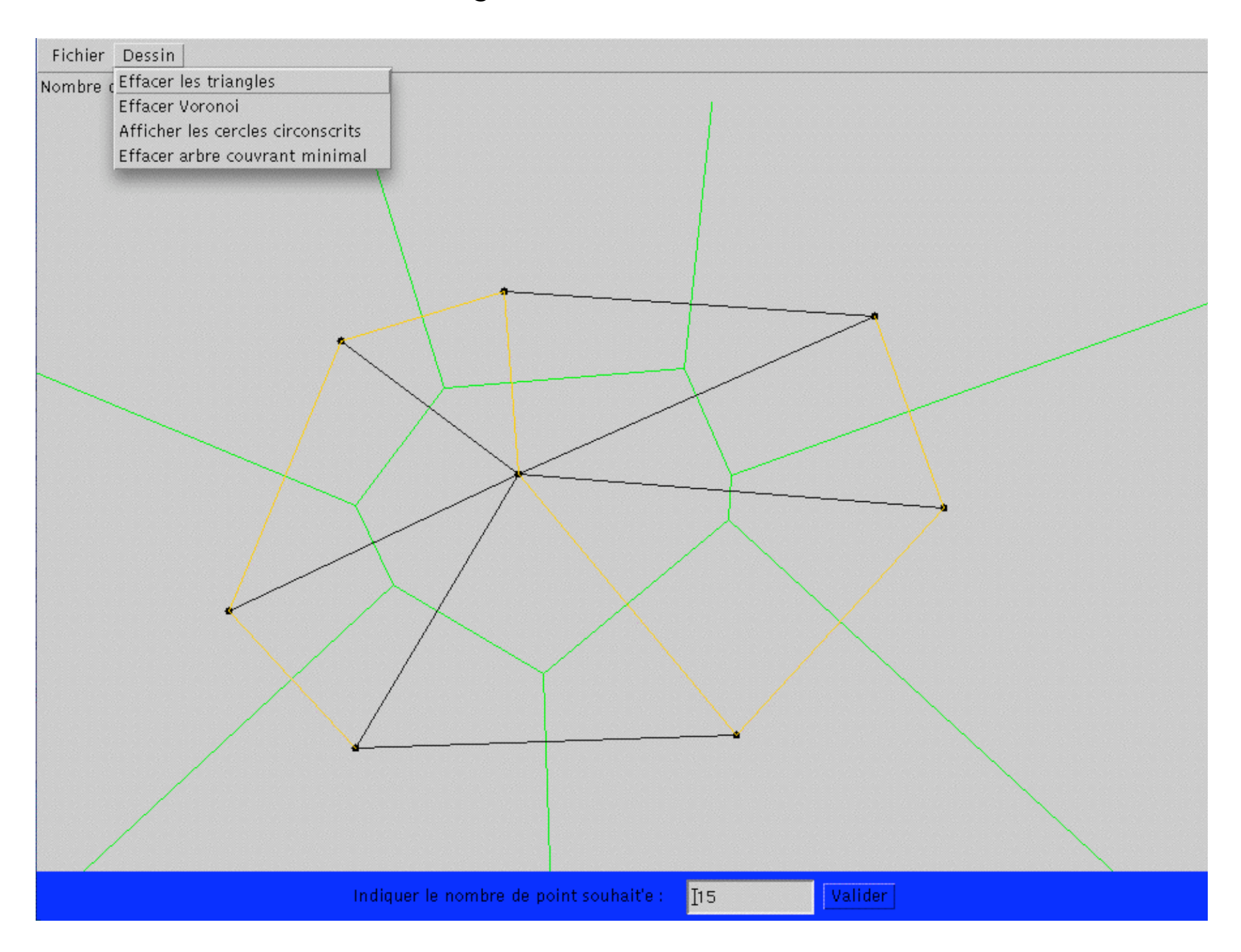
> java Delaunay

Les dimensions de la fenêtre sont alors celles par défaut. Pour les préciser on tape :

> java Delaunay 600 800

pour avoir une fenêtre de hauteur 600 et de largeur 800

En voici une illustration en image :



Vous pouvez constater la présence de menus. Le menu « Dessin » permet de choisir les objets à afficher : triangles, cercles circonscrits, diagramme de Voronoi et arbre couvrant minimal. Quant au menu « Fichier », il permet : de recommencer au début l'insertion de points, de sauvegarder en postscript la triangulation et de fermer le programme.

En bas de la fenêtre, le champ-texte permet de choisir le nombre total de point que l'on souhaite pouvoir insérer. Celui-ci doit être connu au début de la triangulation et ne peut être modifié sous peine d'effacer la triangulation.

Quelques raccourcis clavier :

- « q » permet de quitter le programme
- « n » permet d'effacer la triangulation

Annexe 2 : Listing du code

La classe Point

```
class Point {
   double x ;
   double y ;

   public Point(double x, double y) {
      this.x = x ;
      this.y = y ;
   }

   public String toString() {
      return "x = " + this.x + " y = " + this.y + " :: ";
   }

   public static void main(String[] agrs) {
          double d = Double.NaN ;
          System.out.println(d + (int) d) ;
          System.out.println("d =" + (int) d) ;
   }
}
```

La classe Triangle

```
class Triangle {

   static Point[] points ;
   static int nb ; // correspond au niveau de remplissage du tableau de points
   int i,j,k ;
   Triangle ti,tj,tk ; // les triangles adjacents
   boolean vu, estDessine ;
   Point centre ;

   public Triangle(int i,int j , int k , Triangle ti,Triangle tj, Triangle tk) {
          this.i = i ;
          this.j = j ;
          this.k = k ;
          this.ti = ti ;
          this.tj = tj ;
          this.tk = tk ;
          vu = false ;
          estDessine = false ;
   }

   public Point retourneCentre() {
```

```
        if (centre == null) {
            Point a = points[i] ;
            Point b = points[j] ;
            Point c = points[k] ;
            double X2 = b.x - a.x ;
            double Y2 = b.y - a.y ;
            double Z2 = X2*X2 + Y2*Y2 ;
            double X3 = c.x-a.x ;
            double Y3 = c.y-a.y ;
            double Z3 = X3*X3+Y3*Y3 ;
            double det = X2*Y3-X3*Y2 ;
            double rx = 0.5*(Y3*Z2-Y2*Z3)/det + a.x ;
            double ry = 0.5*(X2*Z3 - X3*Z2)/det + a.y ;
            centre =  new Point(rx,ry) ;
        }
        return centre ;
}

public Point retourneCentreGravite() {
        Point a = points[i] ;
        Point b = points[j] ;
        Point c = points[k] ;
        Point g =new Point((a.x+b.x+c.x)/3,(a.y+b.y+c.y)/3);
        return g;
}

public static int estOriente (int i, int j, int k) {
        // Orientation du triangle 1= direct -1= indirect 0=aligne
        double X2 = points[j].x - points[i].x ;
        double Y2 = points[j].y - points[i].y ;
        double X3 = points[k].x - points[i].x ;
        double Y3 = points[k].y - points[i].y ;
        double det = X2*Y3-X3*Y2;
        if (det>0) return  1;
        if (det<0) return -1;
        return 0;
}

public int dansCercle (int p) {
        // On suppose i, j , k direct
        // 1 si p hors du cercle, 0 sur le cercle, -1 dans le cercle
        double X2 = points[j].x-points[i].x;
        double Y2 = points[j].y-points[i].y;
        double Z2 = X2*X2 + Y2*Y2;
        double X3 = points[k].x-points[i].x;
        double Y3 = points[k].y-points[i].y;
        double Z3 = X3*X3 + Y3*Y3;
        double X4 = points[p].x-points[i].x;
        double Y4 = points[p].y-points[i].y;
        double Z4 = X4*X4 + Y4*Y4;
```

```
        double det = Z2*(X3*Y4-X4*Y3)-Z3*(X2*Y4-X4*Y2)+Z4*(X2*Y3-X3*Y2);
        if (det>0) return  1;
        if (det<0) return -1;
        return 0;
   }

   public boolean dansTriangle (int p) {
        // Attention le triangle doir etre orient'e !!!
        return estOriente(p,i,j)!=-1 && estOriente(p,j,k)!=-1 &&
estOriente(p,k,i)!=-1 ;
   }

   public void print() {
        System.out.println( i + " ; " +j + " ; "+ k ) ;
   }


   public boolean toucheGrandTriangle() {
        return
(i==0)||(i==1)||(i==2)||(j==0)||(j==1)||(j==2)||(k==0)||(k==1)||(k==2);
   }
}
```

La classe Triangulation

```
import java.io.* ;

class Triangulation {

   Triangle triangleInitial ;
   int X_MAX, Y_MAX ;

   public Triangulation(int n, int X_MAX , int Y_MAX) {
        this.X_MAX = X_MAX ;
        this.Y_MAX = Y_MAX ;
        Triangle.points = new Point[n+3] ;
        Triangle.nb = -1 ;
        Triangle.points[0] = new Point(-50*X_MAX,-50*Y_MAX) ; // on choisit un
grand triangle initial pour 'eviter la perte de convexit'e
        Triangle.points[1] = new Point(50*X_MAX,0) ;
        Triangle.points[2] = new Point(0,50*Y_MAX) ;
        triangleInitial = new Triangle(0,1,2,null,null,null) ;
   }

   public void ajouter (Point pt) {
        if (pt.x > X_MAX || pt.y > Y_MAX) {
           System.out.println("Point hors du triangle initial !!!") ;
           return ;
        }
```

```
        if (Triangle.nb+1 < Triangle.points.length) {
           Triangle.nb++ ;
           Triangle.points[Triangle.nb+3] = pt ;
           insere(Triangle.nb+3) ;
        }
   }

   public void insere(int p) {
        Triangle t = localise(p) ;
        Cavite c = extraitCavite(t,p) ;
        remplitCavite(c,p) ;
   }

   public void calcule(Point[] mesPoints, int nombreDePoints) {
        System.out.println("point 0
("+Triangle.points[0].x+","+Triangle.points[0].y+")");
        System.out.println("point 1
("+Triangle.points[1].x+","+Triangle.points[1].y+")");
        System.out.println("point 2
("+Triangle.points[2].x+","+Triangle.points[2].y+")");
        for (int i = 0 ; i < nombreDePoints ;i++) {
           Triangle.points[i+3] = mesPoints[i+3] ;
           Triangle.nb++ ;
        }
        for (int i = 3 ; i < nombreDePoints + 3 ;i++) {
           System.out.println("j'insere le point de coordonnees
("+Triangle.points[i].x+","+Triangle.points[i].y+")");
           insere(i) ;
        }
   }

   public boolean plusPetit(int a, int b, int centre, int depart) {
        //on definit une relation d`ordre sur les points.....
        //retourne true si a<=b, false sinon
        Point pa= Triangle.points[a];
        Point pb= Triangle.points[b];
        Point pcentre= Triangle.points[centre];
        Point pdepart= Triangle.points[depart];
        if ((pa.x==pdepart.x)&&(pa.y==pdepart.y)) return true;
        if ((pb.x==pdepart.x)&&(pb.y==pdepart.y)) return false;
        if (sontOriente(pcentre,pdepart,pa)==sontOriente(pcentre,pdepart,pb))
return sontOriente(pcentre,pa,pb);
      return sontOriente(pcentre,pdepart,pa);
   }



   public Cavite insereDansCavite(int aa,int bb,Triangle tin,Triangle tex,Cavite
cav,int centre) {
        //cette fonction insere un segment dans la cavit'e en respectant l'ordre
```

```
defini par la fonction plusPetit
          //le but est d obtenir au final une cavite form'ee de segment contigus,
tri'e dans le sens direct.
          if (cav==null) {
             System.out.println("j'ai ajoute le segment ["+aa+","+bb+"] dans la
caverne");
             return new Cavite(aa,bb,tin,tex);
          } else return insereDansCaviteAux(aa,bb,tin,tex,cav,centre,cav.a);
   }

   public Cavite insereDansCaviteAux(int aa,int bb, Triangle tin, Triangle tex,
Cavite cav,int centre,int depart) {
          if (cav==null){
             System.out.println("j'ai ajoute le segment ["+aa+","+bb+"] en fin de
caverne");
             return new Cavite(aa,bb,tin,tex);
          }
          if (plusPetit(aa,cav.a,centre,depart)) {
             System.out.println("j'ai ajoute le segment ["+aa+","+bb+"] dans la
caverne");
             return new Cavite (aa,bb,tin,tex,cav);
          } else {
             cav.suivant =
insereDansCaviteAux(aa,bb,tin,tex,cav.suivant,centre,depart);
             return cav;
          }
   }


   public Cavite extraitCavite(Triangle t,int p) {
          Cavite cav=chercheCav(t,p,null);
          return cav;
   }

   public Cavite chercheCav(Triangle t, int p, Cavite cav) {
          t.vu=true;
          System.out.print("je suis dans ");
          t.print();
          Cavite res=cav;
          if (t.ti != null) {
             t.ti.print();
             System.out.println("est  adjacent");
             if (!t.ti.vu){
                    if (!(t.ti.dansCercle(p)==1)) {
                       res = chercheCav(t.ti,p,res) ;
                    } else {
                       res = insereDansCavite(t.j,t.k,t,t.ti,res,p) ;
                    }
             }else {}
          } else {
```

```
            System.out.println("je touche un bord");
            res = insereDansCavite(t.j,t.k,t,t.ti,res,p) ;
        }
        if (t.tj != null) {
            t.tj.print();
            System.out.println("est  adjacent");
            if (!t.tj.vu){
                    if (!(t.tj.dansCercle(p)==1)) {
                        res = chercheCav(t.tj,p,res) ;
                    } else {
                        res = insereDansCavite(t.k,t.i,t,t.tj,res,p) ;
                    }
            }else {}
        } else {
            System.out.println("je touche un bord");
            res = insereDansCavite(t.k,t.i,t,t.tj,res,p) ;
        }
        if (t.tk != null) {
            t.tk.print();
            System.out.println("est  adjacent");
            if (!t.tk.vu){
                    if (!(t.tk.dansCercle(p)==1)) {
                        res = chercheCav(t.tk,p,res) ;
                    } else {
                        res = insereDansCavite(t.i,t.j,t,t.tk,res,p) ;
                    }
            }else {}
        } else {
            System.out.println("je touche un bord");
            res = insereDansCavite(t.i,t.j,t,t.tk,res,p) ;
        }
        System.out.print("fini pour ");
        t.print();
        return res;
    }

    public void remplitCavite(Cavite c, int p) {
        if (c == null) throw new Error("Cavite null ne entr'ee de remplitCavit'e !!!")
;
        c.print() ;
        Triangle to = new Triangle(p,c.a,c.b,c.tExt,null,null) ;
        triangleInitial = to ;
        Triangle dernierCree = to ;
        Triangle courant = to ;
        System.out.print("on cree le triangle ");
        courant.print();
        if (c.tExt!=null) {
            if (c.tExt.ti==c.tInt) c.tExt.ti=courant ;
            else if (c.tExt.tj==c.tInt) {c.tExt.tj=courant;}
            else {c.tExt.tk=courant;}
```

```
            System.out.print("maintenant ");
            c.tExt.print();
            System.out.print("pointe vers ");
            courant.print();
        }
        for (Cavite cs = c.suivant ; cs != null ; cs = cs.suivant ) {
            courant = new Triangle(p,cs.a,cs.b,cs.tExt,null,dernierCree) ;
            System.out.print("on cree le triangle ");
            courant.print();
            if (cs.tExt!=null) {
                    if (cs.tExt.ti==cs.tInt) {cs.tExt.ti=courant;}
                    else if (cs.tExt.tj==cs.tInt) {cs.tExt.tj=courant;}
                    else {cs.tExt.tk=courant;}
                    System.out.print("maintenant ");
                    cs.tExt.print();
                    System.out.print("pointe vers ");
                    courant.print();
            }
            dernierCree.tj = courant ;
            dernierCree = courant ;
        }
        to.tk = courant ;
        courant.tj=to;
    }

    public boolean sontOriente(Point pi, Point pj , Point pk) {
        double X2 = pj.x - pi.x ;
        double Y2 = pj.y - pi.y ;
        double X3 = pk.x - pi.x ;
        double Y3 = pk.y - pi.y ;
        double det = X2*Y3-X3*Y2;
        if (det>=0) return  true;
        else return false ;
    }

    public Triangle localise (int p) {
        System.out.println("D'ebut localise de " + p) ;
        Triangle t = triangleInitial ;
        Point c = t.retourneCentreGravite() ;
        System.out.println("Point de d'epart : " + c) ;
        Point[] points = Triangle.points ;
        while (!t.dansTriangle(p)) {
            //      while (t.dansCercle(p) == 1) {
            if (sontOriente(c,points[t.i],points[p]) &&
sontOriente(c,points[p],points[t.j])) {
                    t = t.tk ;
            } else {
                    if (sontOriente(c,points[t.j],points[p]) &&
sontOriente(c,points[p],points[t.k])) {
                        t = t.ti ;
```

```
                } else {
                    if (sontOriente(c,points[t.k],points[p]) &&
sontOriente(c,points[p],points[t.i])) {
                        t = t.tj ;
                    }
                }
            }
        }
        System.out.print("Triangle localis'e : ") ; t.print() ;
        return t ;
    }

    /************************************************************************/
    /* Fonctions d'impression et d'exportation de la triangulation */
    /************************************************************************/

    public void print() {
        System.out.println("Affichage de la triangulation") ;
        printAux(triangleInitial) ;
        restorePrint(triangleInitial) ;
    }

    public void printAux(Triangle t) {
        if (t != null && !t.vu) {
            t.print() ;
            t.vu = true ;
            printAux(t.ti) ;
            printAux(t.tj) ;
            printAux(t.tk) ;
        }
    }

    public void restorePrint(Triangle t) {
        // permet de remmettre les "vu" `a false
        if (t != null && t.vu) {
            t.vu = false ;
            restorePrint(t.ti) ;
            restorePrint(t.tj) ;
            restorePrint(t.tk) ;
        }
    }

    public String exportPS () {
        // permet d'exporter la triangulation en postscript
        String res = exportPSaux(triangleInitial) ;
        restorePrint(triangleInitial) ;
        res = "%!\n" + res  ;
        return res ;
    }
```

```
public String exportPSaux(Triangle t) {
        String res = "" ;
        if (t != null && !t.vu) {
           t.vu = true ;
           if (!((t.i*t.j*t.k) == 0 || t.i==1 || t.j==1 || t.k==1 || t.i==2 || t.j==2 ||
t.k==2)) {
                if (Delaunay.afficher_triangles) {
                   res = res + "0 0 0 setrgbcolor\n" ;
                   res = res + Triangle.points[t.i].x + " " + Triangle.points[t.i].y + "
moveto " ;
                   res = res + Triangle.points[t.j].x + " " + Triangle.points[t.j].y + "
lineto stroke\n" ;
                   res = res + Triangle.points[t.j].x + " " + Triangle.points[t.j].y + "
moveto " ;
                   res = res + Triangle.points[t.k].x + " " + Triangle.points[t.k].y +
" lineto stroke\n" ;
                   res = res + Triangle.points[t.k].x + " " + Triangle.points[t.k].y +
" moveto " ;
                   res = res + Triangle.points[t.i].x + " " + Triangle.points[t.i].y + "
lineto stroke\n" ;
                }
                if (Delaunay.afficher_cercles) {
                   res = res + "1 0 0 setrgbcolor\n" ;
                   Point c = t.retourneCentre() ;
                   Point a = t.points[t.i] ;
                   double r = Math.sqrt((a.x-c.x)*(a.x-c.x) + (a.y-c.y)*(a.y-c.y)) ;
                   res = res +  c.x + " " +  c.y + " " + (int)  r + " 0 360 arc stroke\n" ;
                }
                if (Delaunay.afficher_voronoi) {
                   res = res + "0 1 0 setrgbcolor\n" ;
                   Point c = t.retourneCentre() ;
                   Point cc ;
                   if (!(t.ti == null)) {
                         cc = t.ti.retourneCentre() ;
                         res = res + c.x + " " + c.y + " moveto " ;
                         res = res + cc.x + " " + cc.y + " lineto stroke\n" ;
                   }
                   if (!(t.tj == null)) {
                         cc = t.tj.retourneCentre() ;
                         res = res + c.x + " " + c.y + " moveto " ;
                         res = res + cc.x + " " + cc.y + " lineto stroke\n" ;
                   }
                   if (!(t.tk == null)) {
                         cc = t.tk.retourneCentre() ;
                         res = res + c.x + " " + c.y + " moveto " ;
                         res = res + cc.x + " " + cc.y + " lineto stroke\n" ;
                   }
                }
           }
           res = res + exportPSaux(t.ti) ;
```

```
            res = res + exportPSaux(t.tj) ;
            res = res + exportPSaux(t.tk) ;
        }
        return res ;
    }


    /******************************************/
    /* construction de l'arbre minimal */
    /******************************************/

    public Arbre retourneArbre() {
        Graphe g = retourneGraphe() ;
        return g.calculerArbre() ;
    }

    public Graphe retourneGraphe() {
        // permet de cr'eer le graphe
        // on ajoute les arÍtes par parcours en profondeur de la triangulation
        Graphe res = new Graphe(Triangle.points, Triangle.nb+1);
        retourneGrapheaux(triangleInitial,res);
        restorePrint(triangleInitial) ;
        return res;
    }

    public void  retourneGrapheaux(Triangle t,Graphe res) {
        t.vu=true;
        res.ajouter(t.j,t.k);
        res.ajouter(t.k,t.i);
        res.ajouter(t.i,t.j);
        if (t.ti != null && t.ti.vu==false)  retourneGrapheaux(t.ti,res);
        if (t.tj != null && t.tj.vu==false)  retourneGrapheaux(t.tj,res);
        if (t.tk != null && t.tk.vu==false)  retourneGrapheaux(t.tk,res);
    }

    public static void main (String[] args) {
        // permet de rentrer les points en input
        // exemple : java Triangulation < input.txt
        // le r'esultat est enregistr'e en postscript dans output.ps
        Lecture.init(System.in) ;
        Point[] mesPoints = Lecture.retourne_points() ;

        Triangulation tn = new Triangulation(mesPoints.length,
Delaunay.LARGEUR, Delaunay.HAUTEUR ) ;
        tn.calcule(mesPoints,mesPoints.length-3) ;
        System.out.println("Fin de la triangulation") ;
        System.out.println("Exportation du graphe") ;
        Graphe g = tn.retourneGraphe() ;
        String filename = "output.ps" ;
        File f = new File (filename);
```

```
        try {
           FileWriter fw = new FileWriter (f);
           // on choisit ce que l'on veut voir affich'e sur le dessin en postscript
           Delaunay.afficher_cercles = false ;
           Delaunay.afficher_voronoi = false ;
           Delaunay.afficher_triangles = true ;
           Delaunay.afficher_arbre = true ;

           String text = tn.exportPS();
           // on ajoute les segments de l'arbre
           System.out.println("Calcul de l'arbre couvrant minimal") ;
           text = text + g.exportArbrePS() ;
           text = text + "showpage" ;
           System.out.println("d'ebut d'ecriture dans "+filename) ;
           int textsize = text.length();
           fw.write (text, 0, textsize);
           fw.close ();
           System.out.println("Enregistrement : " + filename) ;
        } catch (IOException exc) {
           System.out.println("IOException: " + filename);
        }
   }
}
```

La classe Cavite

```
class Cavite {

   int a,b ; // correspondent aux indices des points de l'arÍte de la cavit'e
   Triangle tInt, tExt ;
   // tInt et tExt pointent vers les triangles int'erieurs et ext'erieurs
   Cavite suivant ;
   // Soit c le troisieme point du triangle interieur
   // a et b sont tels que a b c est orient'e

   Cavite (int a, int b ,Triangle tInt, Triangle tExt) {
        this.a = a;
        this.b = b ;
        this.tInt = tInt ;
        this.tExt = tExt ;
   }

   Cavite (int a, int b ,Triangle tInt, Triangle tExt, Cavite suivant) {
        this.a = a;
        this.b = b ;
        this.tInt = tInt ;
        this.tExt = tExt ;
        this.suivant = suivant ;
   }
```

```
    public static Cavite cons (Cavite c1 , Cavite c2) {
            if (c1 == null) return c2 ;
            else {
                Cavite aux = c1 ;
                while (aux.suivant != null) {
                        aux = aux.suivant ;
                }
                aux.suivant = c2 ;
                return c1 ;
            }
    }

    public void print() {
            for (Cavite cs = this ; cs != null ; cs = cs.suivant) {
                System.out.print(cs.a + " " + cs.b);
                if (cs.tExt!=null) {
                System.out.print(" triangle exterieur : ") ;
                cs.tExt.print();
                } else System.out.println(" triangle exterieur null");
            }
    }
}
```

La classe Graphe

```
class Graphe {

    Point[] coords ;
    ListeEntier[] voisins ;
    boolean[][] estDejaAjoute ; // permet d"eviter d'ajouter plusieurs fois la même
arête

    Graphe (Point[] points,int nb) {
            voisins = new ListeEntier[nb] ;
            coords = new Point[nb] ;
            int n = coords.length ;
            for (int i = 0 ; i < n ; i++) {
                coords[i] = points[i+3] ;
            }
            estDejaAjoute = new boolean[nb][nb] ;
    }

    public void ajouter(int i, int j) {
            if (i>2 && j>2 && !estDejaAjoute[i-3][j-3]) {
                estDejaAjoute[i-3][j-3] = true ;
                estDejaAjoute[j-3][i-3] = true ;
                voisins[i-3] = new ListeEntier(j-3,voisins[i-3]) ;
                voisins[j-3] = new ListeEntier(i-3,voisins[j-3]) ;
            }
```

```
    }

    public Arbre calculerArbre() {
            int i = 1 ;
            int n = 1 ; // correspond au nb de points d'ej'a ajout'es
            QueueListe q = new QueueListe(coords.length,coords) ;
            q.ajouter(0,voisins) ;
            Segment s ;
            Arbre res = null ;
            while (n < coords.length  && q.liste != null) {
               s = q.retourneMinimum() ;
               if (q.estAjoute(s.a) && !q.estAjoute(s.b)) {
                       q.ajouter(s.b,voisins) ;
               } else {
                       if (!q.estAjoute(s.a) && q.estAjoute(s.b)) {
                       q.ajouter(s.a,voisins) ;
                       } else throw new Error("segment min ÈrronÈ dans calculerArbre") ;
               }
               res = new Arbre(s,res) ;
               n++ ;
            }
            return res ;
    }

    public String exportArbrePS () {
            // permet d'exporter l'arbre en postscript
            Arbre arb = calculerArbre() ;
            String res = "1 0.7 0 setrgbcolor\n" ;
            for (Arbre aa = arb ; aa!= null ; aa = aa.suite) {
               res = res  + coords[aa.val.a].x + " " + coords[aa.val.a].y + " moveto " ;
               res = res + coords[aa.val.b].x + " " + coords[aa.val.b].y + " lineto
stroke\n " ;
            }
            return res ;
    }
}
```

La classe Arbre

```
class Arbre {
    // stocke les arÍtes de l'arbre comme une liste de segments
    Segment val ;
    Arbre suite ;

    Arbre(Segment s, Arbre suite) {
            val = s ;
            this.suite = suite ;
    }

    void print() {
```

```
        for ( Arbre a = this ; a != null ; a = a.suite) {
            a.val.print() ;
        }
    }
}
```

La classe QueueListe

```
class QueueListe {

    boolean[] s1 ;
    ListeSegment liste ;
    Point[] coords ;

    QueueListe (int n, Point[] coords) {
            s1 = new boolean[n] ;
            this.coords = coords ;
    }

    boolean estAjoute(int i) {
            return s1[i] ;
    }

    void ajouter(int i, ListeEntier[] voisins) {
            s1[i] = true ;
            // on enl`eve les arÍtes contenant i dans la liste
            supprimer(i) ;
            // on rajoute les segments
            int j ;
            for (ListeEntier ll = voisins[i] ; ll != null ; ll = ll.suite) {
                if (s1[ll.val] == false) {
                        j = ll.val ;
                        double d =
                        (coords[i].x - coords[j].x)*(coords[i].x - coords[j].x)+
                        (coords[i].y - coords[j].y)*(coords[i].y - coords[j].y) ;
                        ajouterSegment(new Segment(i,j,d)) ;
                }
            }
    }

    void ajouterSegment(Segment s) {
            liste = ListeSegment.ajouter(liste,s) ;
    }

    void supprimer(int i) {
            liste = supprimerAux(liste,i) ;
    }

    static ListeSegment supprimerAux (ListeSegment ll , int i) {
            if ( ll == null ) return null ;
```

```
        else {
           if (ll.val.a == i || ll.val.b == i) {
                return supprimerAux(ll.suite,i) ;
           } else {
                return new ListeSegment(ll.val,supprimerAux(ll.suite,i)) ;
           }
        }
   }

   Segment retourneMinimum() {
        // renvoie le segment de longueur minimum dont celui `a ajouter
        if (liste == null) throw new Error("Queue vide : impossible de retourner un
minimum") ;
        return liste.val ;
   }
}
```

La classe ListeSegment

```
class ListeSegment {
   // liste croissante de segments
   Segment val ;
   ListeSegment suite ;

   ListeSegment(Segment val) {
        this.val = val ;
   }

   ListeSegment(Segment val, ListeSegment suite) {
        this.val = val ;
        this.suite = suite ;
   }

   public static ListeSegment ajouter(ListeSegment ll , Segment s) {
        // cette m'ethode permet d'ajouter en maintenant le liste tri'ee
        if (ll == null) return new ListeSegment(s) ;
        if (s.longueur < ll.val.longueur) return new ListeSegment(s,ll) ;
        else {
           return new ListeSegment(ll.val, ajouter(ll.suite,s)) ;
        }
   }
}
```

La classe Segment

```
class Segment {
    int a;
    int b;
    double longueur ;
```

```
    Segment(int aa,int bb,double d){
            a=aa;
            b=bb;
            longueur = d ;
    }

    void print() {
             System.out.println(a + " " + b + " de longueur " + longueur) ;
    }
}
```

La classe ListeEntier

```
class ListeEntier {

    int val ;
    ListeEntier suite ;

    ListeEntier(int val , ListeEntier suite ) {
            this.val = val ;
            this.suite = suite ;
    }
}
```

La classe Delaunay

```
import java.awt.* ;
import java.awt.event.* ;
import java.io.* ;

class Delaunay extends Frame implements  ActionListener {

    static int HAUTEUR = 600 ;
    static int LARGEUR = 800 ;

    // nombre de points autoris'es

    static int NB_POINTS = 15 ;

    // champs

    Triangulation triangulation ;
    CanvasDelaunay monCanvas ;
    PanelDelaunay monPanel ;
    Label monLabel ;

    // les booleans n'ecessaires pour l'affichage
    static boolean afficher_voronoi = true ;
    static boolean afficher_cercles = true ;
```

```
static boolean afficher_triangles = true ;
static boolean afficher_arbre = true ;

// d'eclaration de la police
private Font font = new Font("serif", Font.ITALIC+Font.BOLD, 36) ;

// Declarations des menus
static final MenuBar mainMenuBar = new MenuBar();

protected Menu fileMenu;
protected MenuItem miNew;
protected MenuItem miClose;
protected MenuItem miSaveAs;

protected Menu drawMenu;
protected MenuItem miTriangles;
protected MenuItem miVoronoi;
protected MenuItem miCircles;
protected MenuItem miTree;


public void addFileMenuItems() {
   miNew = new MenuItem ("Nouveau");
   fileMenu.add(miNew).setEnabled(true);
   miNew.addActionListener(this);

   miSaveAs = new MenuItem ("Enregistrer sous");
   fileMenu.add(miSaveAs).setEnabled(true);
   miSaveAs.addActionListener(this);

   miClose = new MenuItem ("Fermer");
   fileMenu.add(miClose).setEnabled(true);
   miClose.addActionListener(this);

   mainMenuBar.add(fileMenu);
}


public void addEditMenuItems() {

       if (afficher_triangles) {
   miTriangles = new MenuItem("Effacer les triangles");
       } else   miTriangles = new MenuItem("Afficher les triangles");
   drawMenu.add(miTriangles).setEnabled(true);
   miTriangles.addActionListener(this);

       if (afficher_voronoi) {
   miVoronoi = new MenuItem("Effacer Voronoi");
       } else miVoronoi = new MenuItem("Afficher Voronoi");
   drawMenu.add(miVoronoi).setEnabled(true);
```

```
    miVoronoi.addActionListener(this);

        if (afficher_cercles) {
    miCircles = new MenuItem("Effacer les cercles circonscrits");
        } else miCircles = new MenuItem("Afficher les cercles circonscrits");
    drawMenu.add(miCircles).setEnabled(true);
    miCircles.addActionListener(this);

        if (afficher_arbre) {
    miTree = new MenuItem("Effacer arbre couvrant minimal");
        } else miTree = new MenuItem("Afficher arbre couvrant minimal");
    drawMenu.add(miTree).setEnabled(true);
    miTree.addActionListener(this);

    mainMenuBar.add(drawMenu);
}

public void addMenus() {
    fileMenu = new Menu("Fichier");
    drawMenu = new Menu("Dessin");
    addFileMenuItems();
    addEditMenuItems();
    setMenuBar (mainMenuBar);
}

public Delaunay() {
    super("");
    WindowAdpt WAdapter = new WindowAdpt();
    this.addWindowListener(WAdapter);

    setTitle ("Delaunay");
        setResizable(false) ;
    setLayout(null);
    addMenus();

    setLayout(new BorderLayout());

        triangulation = new Triangulation(NB_POINTS, LARGEUR, HAUTEUR) ;

        monLabel = new Label("Veuillez saisir les points \'a la souris.") ;
        this.add("North",monLabel) ;
        monCanvas = new CanvasDelaunay(triangulation , monLabel) ;
    this.add("Center",monCanvas) ;

    monPanel = new PanelDelaunay(triangulation, monCanvas) ;
    this.add("South",monPanel) ;

    this.setSize(LARGEUR,HAUTEUR) ;
    setVisible(true);
```

```
    }

    public void handleQuit()
    {
       System.exit(0);
    }

    // ActionListener pour les menus
    public void actionPerformed(ActionEvent newEvent)
    {
       if (newEvent.getActionCommand().equals(miNew.getActionCommand()))
doNew();
       else if (newEvent.getActionCommand().equals(miClose.getActionCommand()))
doClose();
       else if
(newEvent.getActionCommand().equals(miSaveAs.getActionCommand()))
doSaveAs();
       else if
(newEvent.getActionCommand().equals(miTriangles.getActionCommand()))
doTriangles();
       else if
(newEvent.getActionCommand().equals(miVoronoi.getActionCommand()))
doVoronoi();
       else if
(newEvent.getActionCommand().equals(miCircles.getActionCommand()))
doCircles();
       else if (newEvent.getActionCommand().equals(miTree.getActionCommand()))
doTree();
    }

    public void doNew() {
            triangulation = new Triangulation(NB_POINTS, LARGEUR, HAUTEUR) ;
            monCanvas.redessiner() ;
    }

    public void doClose() {
            handleQuit() ;
    }

    public void doSaveAs() {
            //          triangulation.print() ;
            FileDialog file = new FileDialog (Delaunay.this, "Sauver dessin",
FileDialog.SAVE);
            file.show(); // Blocks
            String curFile;
            if ((curFile = file.getFile()) != null) {
               String filename = file.getDirectory() + curFile;
               setCursor (Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
               String aux = "" ;
               if (Delaunay.afficher_arbre) {
```

```
                Graphe g = triangulation.retourneGraphe() ;
                aux = g.exportArbrePS() ;
            }
            File f = new File (filename);
            try {
                FileWriter fw = new FileWriter (f);
                String text = triangulation.exportPS();
                text = text + aux + "showpage" ;
                int textsize = text.length();
                fw.write (text, 0, textsize);
                fw.close ();
                System.out.println("Enregistrement : " + filename) ;
            } catch (IOException exc) {
                System.out.println("IOException: " + filename);
            }
            setCursor (Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));
        }
}

public void doVoronoi() {
        if (afficher_voronoi) {
            System.out.println("Fin d'affichage de Voronoi") ;
            miVoronoi.setLabel("Afficher Voronoi") ;
            afficher_voronoi = false ;
        } else {
            System.out.println("affichage de Voronoi") ;
            miVoronoi.setLabel("Effacer Voronoi") ;
            afficher_voronoi = true ;
        }
        monCanvas.repaint() ;
}

public void doCircles() {
        if (afficher_cercles) {
            System.out.println("Fin d'affichage des cercles") ;
            miCircles.setLabel("Afficher les cercles circonscrits") ;
            afficher_cercles = false ;
        } else {
            System.out.println("affichage des cercles") ;
            miCircles.setLabel("Effacer les cercles circonscrits") ;
            afficher_cercles = true ;
        }
        monCanvas.repaint() ;
}

public void doTriangles() {
        if (afficher_triangles) {
            System.out.println("Fin d'affichage des triangles") ;
            miTriangles.setLabel("Afficher les triangles") ;
            afficher_triangles = false ;
```

```
        } else {
            System.out.println("affichage des triangles") ;
            miTriangles.setLabel("Effacer les triangles") ;
            afficher_triangles = true ;
        }
        monCanvas.repaint() ;
    }

    public void doTree() {
        if (afficher_arbre) {
            System.out.println("Fin d'affichage de l'arbre") ;
            miTree.setLabel("Afficher l'arbre couvrant minimal") ;
            afficher_arbre = false ;
        } else {
            System.out.println("affichage de l'arbre") ;
            miTree.setLabel("Effacer l'arbre couvrant minimal") ;
            afficher_arbre = true ;
        }
        monCanvas.repaint() ;
    }

    class WindowAdpt extends java.awt.event.WindowAdapter {
        public void windowClosing(java.awt.event.WindowEvent event) {
            handleQuit();
        }
    }

    public static void main(String args[]) {
        if (args.length == 2) {
            Delaunay.HAUTEUR = Integer.parseInt(args[0]) ;
            Delaunay.LARGEUR = Integer.parseInt(args[1]) ;
        }
        new Delaunay();
    }
}
```

La classe CanvasDelaunay

```
import java.awt.* ;
import java.awt.event.* ;

class CanvasDelaunay extends Canvas implements MouseListener {

    Triangulation t ;
    Label l ;
    boolean clear = false ;

    CanvasDelaunay(Triangulation t, Label l) {
        this.t = t ;
```

```
        this.l = l ;
        addMouseListener(this);
        addKeyListener(new MyKeyAdapter(this)) ;
    }

    // Methode appel'ee pour red'essiner tout le Canvas
    public void paint(Graphics g) {
        if (!clear) {
            dessiner(g);
        }
    }

    public void dessiner(Graphics g) {
        g.setColor(Color.BLACK) ;
        for (int i = 3; i < Triangle.nb+1+3 ; i++) {
            g.fillOval((int) Triangle.points[i].x - 3 , (int) Triangle.points[i].y - 3 , 6 ,
6) ;
        }
        if (Delaunay.afficher_voronoi) { dessinerVoronoi(t.triangleInitial , g) ;
restore(t.triangleInitial) ; } ;
        if (Delaunay.afficher_cercles) { dessinerCercles(t.triangleInitial , g) ;
restore(t.triangleInitial) ; } ;
        if (Delaunay.afficher_triangles) { dessinerTriangles(t.triangleInitial , g) ;
restore(t.triangleInitial) ; } ;
        if (Delaunay.afficher_arbre) { dessinerArbre(g) ; }
    }

    public void dessinerArbre(Graphics g) {
        // code ‡ insÈrer pour dessiner l'arbre
        if (Triangle.nb > 1) {
            g.setColor(Color.ORANGE) ;
            if (Delaunay.afficher_arbre) {
                Arbre a = t.retourneArbre() ;
                for (Arbre aa = a ; aa != null ; aa = aa.suite) {
                    Point p = Triangle.points[aa.val.a+3];
                    Point q = Triangle.points[aa.val.b+3];
                    g.drawLine((int) p.x, (int) p.y , (int) q.x , (int) q.y) ;
                }
            }
        }
    }

    public void dessinerVoronoi(Triangle tt , Graphics g) {
        g.setColor(Color.GREEN) ;
        if (tt == null  || tt.estDessine == true ) return ;
        tt.estDessine = true ;
        Point c = tt.retourneCentre() ;
        if (!(tt.i == 0 || tt.j == 0 || tt.k == 0 || tt.i == 1 || tt.j == 1 || tt.k == 1
|| tt.i == 2 || tt.j == 2 || tt.k == 2)) {
            if (tt.ti != null) {
```

```
                Point ci = tt.ti.retourneCentre() ;
                g.drawLine((int) c.x , (int) c.y , (int) ci.x , (int) ci.y) ;
            }
            if (tt.tj != null) {
                Point cj = tt.tj.retourneCentre() ;
                g.drawLine((int) c.x , (int) c.y , (int) cj.x , (int) cj.y) ;

            }
            if (tt.tk != null) {
                Point ck = tt.tk.retourneCentre() ;
                g.drawLine((int) c.x , (int) c.y , (int) ck.x , (int) ck.y) ;
            }
        }
        if (tt.ti != null) { dessinerVoronoi(tt.ti, g) ; }
        if (tt.tj != null) { dessinerVoronoi(tt.tj, g) ; }
        if (tt.tk != null) { dessinerVoronoi(tt.tk, g) ; }
    }

    public void dessinerCercles(Triangle tt , Graphics g) {
        if (tt == null  || tt.estDessine == true ) return ;
        g.setColor(Color.RED) ;
        tt.estDessine = true ;
        if (!(tt.i == 0 || tt.j == 0 || tt.k == 0 || tt.i == 1 || tt.j == 1 || tt.k == 1
|| tt.i == 2 || tt.j == 2 || tt.k == 2)) {
        Point c = tt.retourneCentre() ;
        Point a = tt.points[tt.i] ;
        double r = Math.sqrt((a.x-c.x)*(a.x-c.x) + (a.y-c.y)*(a.y-c.y)) ;
        g.drawOval((int) (c.x-r) , (int) (c.y -r) , (int) (2*r), (int) (2*r)) ;
        }
        dessinerCercles(tt.ti, g) ;
        dessinerCercles(tt.tj, g) ;
        dessinerCercles(tt.tk, g) ;
    }

    public void dessinerTriangles(Triangle tt , Graphics g) {
        if (tt == null  || tt.estDessine == true ) return ;
        g.setColor(Color.BLACK) ;
        tt.estDessine = true ;
        if (!(tt.i == 0 || tt.j == 0 || tt.k == 0 || tt.i == 1 || tt.j == 1 || tt.k == 1
|| tt.i == 2 || tt.j == 2 || tt.k == 2)) {
            Point a = tt.points[tt.i] ;
            Point b = tt.points[tt.j] ;
            Point c = tt.points[tt.k] ;
            g.drawLine((int) a.x , (int) a.y , (int) b.x , (int) b.y) ;
            g.drawLine((int) b.x , (int) b.y , (int) c.x , (int) c.y) ;
            g.drawLine((int) c.x , (int) c.y , (int) a.x , (int) a.y) ;
        }
        dessinerTriangles(tt.ti, g) ;
        dessinerTriangles(tt.tj, g) ;
        dessinerTriangles(tt.tk, g) ;
```

```
    }

    public void restore(Triangle tt) {
            if (tt != null && tt.estDessine) {
               tt.estDessine = false ;
               restore(tt.ti) ;
               restore(tt.tj) ;
               restore(tt.tk) ;
            }
    }

    public void redessiner() {
        clear = true ;
            l.setText("Veuillez saisir les points 'a la souris.") ;
            t = new Triangulation(Delaunay.NB_POINTS, Delaunay.LARGEUR,
Delaunay.HAUTEUR) ;
        repaint() ;
    }

    // Liste des methodes du MouseListener
    public void mouseClicked(MouseEvent e) {
            if (Triangle.nb + 1 >= Delaunay.NB_POINTS) {
               l.setText("Triangulation termin'ee !") ;
            } else {
               int x = e.getX() ;
               int y = e.getY() ;
               clear = false ;
               Point pt = new Point((double) x , (double) y) ;
               System.out.println("Ajout de " +  pt) ;
               t.ajouter(pt) ;
               l.setText("Nombre de points restants `a placer : " +
(Delaunay.NB_POINTS - Triangle.nb - 1)) ;
            }
            repaint() ;
    }

    public void mousePressed(MouseEvent e) {}

    public void mouseReleased(MouseEvent e) {}

    public void mouseEntered(MouseEvent e) {
            requestFocus();
            }

    public void mouseExited(MouseEvent e) {}

    public void nouveau() {
            t = new Triangulation(Delaunay.NB_POINTS, Delaunay.LARGEUR,
Delaunay.HAUTEUR) ;
            redessiner() ;
```

```
    }
}
```

La classe MyKeyAdapter

```
class MyKeyAdapter extends KeyAdapter {

    CanvasDelaunay c ;

    public MyKeyAdapter(CanvasDelaunay c) {
            this.c = c ;
    }

    public void keyTyped(KeyEvent e) {
            switch ( e.getKeyChar() ) {
            case ('q') : System.exit(0) ;
            case ('n') : { c.nouveau() ; break ;}
            default : System.out.println("Caractere inconnu "+ e.getKeyChar() ) ;
break ;
            }
    }
}
```

La classe PanelDelaunay

```
import java.awt.* ;
import java.awt.event.* ;

class PanelDelaunay extends Panel implements ActionListener {

    CanvasDelaunay monCanvas ;
    Triangulation t ;

    TextField monTextField ;
    Button boutonValider;

    public PanelDelaunay(Triangulation t, CanvasDelaunay monCanvas) {
            this.monCanvas = monCanvas ;
            this.t = t ;

        setBackground(Color.blue);
            add(new Label("Indiquer le nombre de point souhait'e :")) ;
            monTextField = new TextField(Integer.toString(Delaunay.NB_POINTS),10) ;
            add(monTextField) ;
        add(boutonValider = new Button("Valider"));
        boutonValider.addActionListener(this);
    }
```

```
    public void actionPerformed(ActionEvent ev) {
        String label = ev.getActionCommand();

        if (label.equals("Valider")) {
                int n = Integer.parseInt(monTextField.getText()) ;
                Delaunay.NB_POINTS = n ;
                t = new Triangulation(n, Delaunay.LARGEUR, Delaunay.HAUTEUR) ;
                monCanvas.redessiner() ;
        }
    }
}
```

La classe Lecture

```
import java.io.* ;

class Lecture {

    // ------------ lecture de l'entr'ee ------------------

    // Variable statique li'ees `a la lecture par avancer() :
    static char carCourant ; // caract`ere courant
    static BufferedReader in ; // Stream de lecture

    final static char carEOF = (char) -1 ; // caract`ere codant la fin de fichier

    static void avancer () {
            try {
                carCourant = (char) in.read () ;
                // read() renvoie un int.
            } catch (java.io.IOException e) {
                // En cas d'exception on consid`ere la fin de fichier
            //  atteinte (cod'ee par -1 selon read() ) :
                carCourant = carEOF ;
            }
    }

    static void init (InputStream inp) {
            // par exemple inp = System.in
            in = new BufferedReader (new InputStreamReader (inp)) ;
            avancer () ;
    }


    // ------------ lecture de in -----------------

    static Point[] retourne_points() {
            ListePoints lp = null  ;
            int nb = 0 ;
```

```
        while (carCourant != carEOF) {
           nb++ ;
           lp = new ListePoints(point_suivant() ,lp) ;
        }
        Point[] res = new Point[nb+3] ;
        nb = 0 ;
        for (ListePoints ll = lp ; ll != null ; ll = ll.suivant) {
           res[nb+3] = ll.val ;
           nb++ ;
        }
        return res ;
  }

  static Point point_suivant() {
        lireBlancs() ;
        double x = nombre() ;
        lireBlancs() ;
        double y = nombre() ;
        lireBlancs() ;
        return new Point(x,y) ;
  }

  // Lecture d'un nombre :
  static private double nombre () {
        if (! Character.isDigit (carCourant)) throw new Error("Un chiffre est
attendu en entrÈe de nombre() !!!") ;
        double n = 0 ;
        while (Character.isDigit (carCourant)) {
           int chiffre = ((int)carCourant) - ((int)'0') ;
           // Les caract`eres 0,1,2,... sont cod'es dans cet ordre
           //   en ascii par des nombres cons'ecutifs (voir man ascii).
           n = n * 10 + chiffre ;
           avancer () ;
        }
        if (carCourant == '.') {
           avancer() ;
           int r  = 10 ;
           while (Character.isDigit (carCourant)) {
                 int chiffre = ((int)carCourant) - ((int)'0') ;
                 // Les caract`eres 0,1,2,... sont cod'es dans cet ordre
                 //   en ascii par des nombres cons'ecutifs (voir man ascii).
                 n = n + chiffre/r ;
                 r = r*10 ;
                 avancer () ;
           }
        }
        return (double) n ;
  }

  // Lecture des blancs :
```

```
    static void lireBlancs () {
        while (carCourant == ' ' || carCourant == '\t' || carCourant == '\n' )
            avancer () ;
    }

    public static void main (String[] args) throws IOException{
        init (System.in) ; // Initialisation de carCourant.
        Point[] mesPoints = Lecture.retourne_points() ;
        for (int i = 0 ; i < mesPoints.length ; i++) {
            System.out.println(mesPoints[i]) ;
        }
    }
}
```